



## Fog Computing

NUNO TIAGO MELO GONÇALVES

Outubro de 2019

# Fog Computing

**Nuno Tiago Melo Gonçalves**

**Advisor: Carlos Ferreira**

Porto, 15<sup>th</sup> October 2019



# Abstract

Everything that is not a computer, in the traditional sense, is being connected to the Internet. These devices are also referred to as the Internet of Things and they are pressuring the current network infrastructure. Not all devices are intensive data producers and part of them can be used beyond their original intent by sharing their computational resources. The combination of those two factors can be used either to perform insight over the data closer where is originated or extend into new services by making available computational resources, but not exclusively, at the edge of the network. Fog computing is a new computational paradigm that provides those devices a new form of cloud at a closer distance where IoT and other devices with connectivity capabilities can offload computation.

In this dissertation, we have explored the fog computing paradigm, and also comparing with other paradigms, namely cloud, and edge computing. Then, we propose a novel architecture that can be used to form or be part of this new paradigm. The implementation was tested on two types of applications. The first application had the main objective of demonstrating the correctness of the implementation while the other application, had the goal of validating the characteristics of fog computing.

**Keywords:** Fog Computing; Cloud Computing; Edge Computing; Offloading Computing



# Resumo

Tudo o que não é um computador, no sentido tradicional, está sendo conectado à Internet. Esses dispositivos também são chamados de Internet das Coisas e estão pressionando a infraestrutura de rede atual. Nem todos os dispositivos são produtores intensivos de dados e parte deles pode ser usada além de sua intenção original, compartilhando seus recursos computacionais. A combinação desses dois fatores pode ser usada para realizar processamento dos dados mais próximos de onde são originados ou estender para a criação de novos serviços, disponibilizando recursos computacionais periféricos à rede. Fog computing é um novo paradigma computacional que fornece a esses dispositivos uma nova forma de nuvem a uma distância mais próxima, onde “Things” e outros dispositivos com recursos de conectividade possam delegar processamento.

Nesta dissertação, exploramos fog computing e também comparamos com outros paradigmas, nomeadamente cloud e edge computing. Em seguida, propomos uma nova arquitetura que pode ser usada para formar ou fazer parte desse novo paradigma. A implementação foi testada em dois tipos de aplicativos. A primeira aplicação teve o objetivo principal de demonstrar a correção da implementação, enquanto a outra aplicação, teve como objetivo validar as características de fog computing.

**Palavras-chaves:** Fog Computing; Cloud Computing; Edge Computing; Offloading Computing



# Table of Contents

<b>1</b>	<b><i>Introduction</i></b>	<b>1</b>
1.1	Background	1
1.2	Problem	2
1.3	Objective	2
1.4	Analysis	3
1.5	Methodology	3
1.6	Document Structure	4
<b>2</b>	<b><i>State of Art</i></b>	<b>5</b>
2.1	A New Computing Paradigm	5
2.2	Fog Computing	9
2.2.1	Fog Definition	9
2.2.2	Fog Computing Properties	10
2.2.3	Fog Nodes	11
2.2.4	Fog Models	11
2.2.5	Cloud Computing	12
2.2.6	Edge Computing	13
2.2.7	Edge Computing vs Fog Computing vs Cloud Computing	13
2.3	Fog Middlewares	16
2.3.1	Cloudlets	16
2.3.2	Mobile Edge Computing	18
2.3.3	Micro Data Centres	19
2.3.4	Nano Data Centers	20
2.3.5	Delay Tolerant Networks	21
2.3.6	Femto Clouds	22
2.3.7	Fog middleware comparison	25
2.4	Fog Computing Techniques	26
2.5	Commercial Solutions	32
2.5.1	Cisco I0x	32
2.5.2	FogHorn	34
2.5.3	Big Cloud Providers	35
<b>3</b>	<b><i>Solution Description</i></b>	<b>37</b>
3.1	Vision	37
3.1.1	Positioning	39
3.1.2	Stakeholders Description	39
3.1.3	Product overview	40
3.1.4	Value Analysis	41



<b>3.2</b>	<b>Requirements</b>	<b>42</b>
3.2.1	Domain Model	43
3.2.2	Use Cases	47
3.2.3	Web Platform Use Cases	47
3.2.4	Scheduler Use Cases	50
3.2.5	Runtime Use Cases	54
3.2.6	Supplementary Specification	56
<b>3.3</b>	<b>Technologies</b>	<b>57</b>
3.3.1	Apache Cassandra	57
3.3.2	RabbitMQ	65
3.3.3	Protocol Buffers	69
<b>3.4</b>	<b>Design &amp; Implementation</b>	<b>71</b>
3.4.1	Web Platform	71
3.4.2	Scheduler	87
3.4.3	Runtime	94
3.4.4	Alternative Design & Implementations	97
<b>4</b>	<b><i>Results and Discussion</i></b>	<b>99</b>
4.1.1	TODO Application	99
4.1.2	Motion Detection on Real-Time Video Streaming	112
<b>5</b>	<b><i>Conclusion</i></b>	<b>119</b>
5.1.1	Other Related Works	122
	<b><i>Bibliography</i></b>	<b>124</b>

# List of figures

Figure 1 Fog-Cloud interplay (Bonomi, et al., 2014).....	7
Figure 2 Paradigm shift .....	9
Figure 3 Architecture overview of the new computing model (Iorga, et al., 2018).....	10
Figure 4 Cloudlet functionality interaction (Satyanarayanan, et al., 2009) .....	17
Figure 5 MEC server platform overview (Patel, et al., 2014) .....	18
Figure 6 Fog MDC Overview (Huh, 2015) .....	19
Figure 7 Modem Uptime per ISP (Valancius, et al., 2009).....	20
Figure 8 TCP/IP routing (Warthman, 2015) .....	21
Figure 9 DNT routing (Warthman, 2015).....	22
Figure 10 Femtocloud architecture (Habak, et al., 2015).....	23
Figure 11 Femtocloud numerical results (Habak, et al., 2015).....	24
Figure 12 Numerical results of energy consumption in Zhao's paper (Zhao, et al., 2016). .....	27
Figure 13 SDN-Fog architecture overview (Liang, et al., 2017). .....	28
Figure 14 Aura numerical results (Hasan, et al., 2017).....	29
Figure 15 Lyapunov based algorithm numerical results (Pu, et al., 2016). .....	30
Figure 16 Cisco I0x application life-cycle state diagram (Cisco, 2016). .....	33
Figure 17 Maintenance Types (Hegde, 2018).....	35
Figure 18 Product overview .....	41
Figure 19 Web platform domain model .....	43
Figure 20 Example usage scenario.....	45
Figure 21 Scheduler domain model.....	46
Figure 22 Runtime domain model .....	46
Figure 23 Web platform use cases.....	48
Figure 24 Scheduler use cases .....	51
Figure 25 Runtime use cases .....	55
Figure 26 Cassandra entities.....	58
Figure 27 RabbitMQ messaging overview .....	66
Figure 28 RabbitMQ RCP pattern .....	67
Figure 29 Protocol buffers message definition.....	70
Figure 30 Logic view.....	71
Figure 31 Web platform class diagram .....	72
Figure 32 Lambda Table.....	74
Figure 33 User Resource .....	78
Figure 34 Activation Resource .....	79
Figure 35 Lambda resource .....	80
Figure 36 Form features.....	82
Figure 37 Platform console overview .....	83
Figure 38 Platform project view .....	83
Figure 39 Platform lambda general characteristics form .....	84
Figure 40 Platform lambda function form .....	84

Figure 41 Platform lambda network event .....	85
Figure 42 Execution flow .....	85
Figure 43 Deployment event log .....	86
Figure 44 Logger lambda execution log .....	86
Figure 45 Scheduler class diagram .....	87
Figure 46 REST Deployment entity .....	88
Figure 47 Lambda execution .....	89
Figure 48 HTTP client event trigger .....	90
Figure 49 ProtoBuf Command message definition .....	91
Figure 50 ProtoBuf deploy lambda message definition .....	92
Figure 51 ProtoBuf Invocation and Event message definition .....	93
Figure 52 ProtoBuf event log message definition .....	93
Figure 53 Runtime start up communications .....	94
Figure 54 Runtime class diagram .....	95
Figure 55 Lambda runtime API .....	97
Figure 56 TODO REST API .....	100
Figure 57 Deployment scenario 1: local deployment test .....	101
Figure 58 Deployment scenario 2: fog and cloud deployment test .....	101
Figure 59 Local deployment time spent .....	103
Figure 60 Local deployment throughput test .....	104
Figure 61 Local deployment lambda TODO app CPU usage .....	105
Figure 62 Local deployment standalone app CPU usage .....	106
Figure 63 Local deployment lambda TODO app memory usage .....	106
Figure 64 Local deployment standalone app memory usage .....	107
Figure 65 Fog-cloud deployment time spent .....	108
Figure 66 Fog-cloud deployment throughput test .....	108
Figure 67 Fog-cloud deployment lambda TODO app CPU usage .....	109
Figure 68 Fog-cloud deployment standalone app CPU usage .....	110
Figure 69 Fog-cloud deployment lambda TODO app memory usage .....	110
Figure 70 Fog-cloud deployment standalone app memory usage .....	111
Figure 71 Background subtraction .....	113
Figure 72 Video application execution time .....	114
Figure 73 Cloud video analysis rate .....	116
Figure 74 Fog video analysis rate .....	117

# List of Tables

Table 1 Cloud and Fog computing key differences.....	14
Table 2 Fog and Edge key differences .....	15
Table 3 Fog middleware’s resume .....	25
Table 4 Workload & Scalability middleware’s criteria.....	26
Table 5 Cloud providers and their IoT solutions. ....	35
Table 6 High level goals.....	40
Table 7 Cassandra compaction strategies .....	60
Table 8 Weak vs strong consistency systems .....	61
Table 9 Cassandra consistency levels .....	62
Table 10 Primary keys declarations .....	64
Table 11 Partition key behavior .....	65
Table 12 Variable integer encoding .....	69
Table 13 Database tables.....	76
Table 14 Platform REST resources summary .....	80



# Acronyms and Symbols

AI – Artificial Intelligence  
API – Application Program Interface  
CCTV – Closed-Circuit Television  
DC – Datacenter  
DNN – Deep Neural Network  
DNS – Domain Name System  
EAV– Entity Attribute Value  
FLOPS – Floating Point Operations per Second  
FPS – Frame per Second  
HTML – Hypertext Markup Language  
HTTP – Hypertext Transfer Protocol  
IoT – Internet of Things  
IoT – Internet of Things  
IP – Internet Protocol  
ISP – Internet Service Provider  
IT – Information Technology  
JSON – JavaScript Object Notation  
JVM – Java Virtual Machine  
LAN – Local Area Network  
LIFI – Light Fidelity  
LTE – Long Term Evolution  
MTM – Machine to Machine  
MTU – Maximum Transmission Unit  
NBI – North Bound Interface  
NoSQL – Not Only Structured Query Language  
OPEX – Operating expense, operating expenditure, operational expense, operational expenditure  
OS – Operative System  
P2P – Peer to Peer  
PCL – Programmable Logic Controller  
PDU – Protocol Data Unit  
QoE- Quality of Experience  
QoS – Quality of Service  
REST – Representational State Transfer  
RPC – Remote Procedure Call  
SIMD – Single Instruction Multiple Data  
SQL – Structured Query Language  
TCP – Transport Control Protocol  
VM – Virtual Machine  
WAN – Wide Area Network  
XML – Extensible Markup Language



# 1 Introduction

This chapter introduces the reader to the theme in discussion. We start to highlight some aspects that brought fog computing into existence. Then, we define the problem and with that, we defined the objectives for this work. We conclude with the analysis approach followed by the document structure.

## 1.1 Background

Technology have contributed for the civilization development in many areas. In communication, we have the power of information at distance of our fingerprint at anytime, anywhere, and that, was just a dream of mid 1990s (Satyanarayanan, et al., 2009). The volume of resources to empower this reality have increased accordingly. Among them, one resource is electricity. The consumption has increased since his commercialization, and there is no indication that will stabilize over the futures years (EnerData, 2018).

Energy is needed to power residential homes, companies, data centers, cell towers, to charge the battery of millions of transportable devices, and so on. The IT area strives to design energy-friendly solutions. Google, for example, increased the baseline temperature in 4 degrees over their data centers (Alphabet, 2018), another recently attempt was to use Artificial Intelligence to maximize cooling efficiency (Evans & Gao, 2016).

Aside the ambient benefits of achieve greater energy efficiency, for the energy market perspective, refining it by 1% in fuel saving over 15 years can represent 58 billions of euros (Evans & Annunziata, 2012). The power of just 1% is pointed as achievable by having intelligent and advanced analytics.



In today's world, we have powerful mathematical analytics tools, plus, the machinery, which is progressive being equipped with or more computational resources, creates a scenario where large quantities of data are being produced, ready to be digitally processed to create more complete mathematical models, pushing towards the Industry 4.0.

This machinery, also known as IoT devices, in 2016 generated 280 Zettabytes (Cisco, 2016), and the trend is to increase, plus, is not just the data, is the number of connected devices as well. Moving alone all the data to perform analytics has become a challenge under the current cloud model. AWS from Amazon, already have extreme solutions such as trucks (snowmobile) equipped with trailers to move petabytes of data (Amazon, 2019). Amazon makes available this solution to help companies to move their data over to AWS cloud infrastructure due to the Internet bandwidth being insufficient to move data on time.

Analytics is changing due to the massive data that cannot be transferred over the network, and extreme commercial solutions are not for every company, so, moving the analytics closer to data is the rational alternative (Enescu, 2014). This new model places the intelligence closer/over the nodes that once were only data producers, bringing the 3<sup>rd</sup> wave of a new computational model, where is prevalent machine to machine M2M communication over a highly decentralized architecture.

## **1.2 Problem**

Devices are being equipped with more features offering computing, connectivity and storage. In numbers they are also rising. Most of them already require Internet access to perform several tasks, such as send us push notifications via a companion device app, or simply to report back to the manufacturer about its state. The volume of data generated is pressuring Wide Area Networks. The data push model has its costs: as latency for regular users, or real money for data-heavy enterprises. Another aspect is resource waste - most of those devices require electricity 24/7, staying idle most of the time. That waste can be converted to perform tasks that other networked devices are required to perform, such as running heavy algorithms.

## **1.3 Objective**

In theory, this work has the purpose of exposing fog computing, what are the reasons of its existence, which characteristics differentiate fog, edge and cloud computing and how this paradigm connects edge computing and cloud computing.

Create a platform to study the fog paradigm properties, combining edge computing and cloud computing. In the platform that was developed, we have run a set of typical applications: i) motion detection over a video stream, and an ii) a generic REST application. Both test applications were executed at cloud and fog context and from the different scenarios, we conduct a resource consumption differential analysis.

## **1.4 Analysis**

The resource consumption differential analysis was performed in terms of computational, network resources. The applications were deployed in both environments, fog, and cloud, where the REST application was also used to test the platform implementation and how it performs under benchmark tests. The video stream application besides the computational resource's analysis, we also reasoning about the video stream application computational and monetary requirements to deploy such systems.

## **1.5 Methodology**

The methodology can be separated into two phases: i) investigation and ii) development. During both phases, we had non-formal meetings with the dissertation advisors to show and questioning the progress that has been made since the last meeting. In the investigation phase, we have adopted the agile methodology, reviewing what was written and planning what will be done, in periods of two weeks long.

In the development phase, we have adopted the waterfall methodology, where the requirements took extra attention to end this dissertation with a functional, ready to be tested and to be used. The waterfall was adopted by two reasons: i) complex solution, and ii) time constraint. The complexity and the impulse of adding and improving pre-defined features lead to the adoption of this model since we begin with a set of well-defined features and all our attention and afford was to complete and test those features. The code is versioned controlled and is available for public use as also the platform, that is under this [link](#).

## **1.6 Document Structure**

This work is constituted by 5 chapters: i) introduction, ii) state of art, iii) solution description, iv) design and v) conclusion. Each chapter contains various section and subsections structured in hierarchal order.

In introduction we present to the reader the overall problem and reasoning for pursuing fog computing. We then offer a detailed review of the most important concepts and state-of-the-art technologies and solutions available. Such review is followed by putting forward our vision and main arquitectural characteristics of an implementation of the concept. Finally, we discuss the results that were obtained in two different scenarios and argue for a conclusion on the relative merits and shortcomings of fog computing technology.

## 2 State of Art

In this chapter is where it is defined fog computing in their properties according to the literature, comparing it against existing computing models. Then describe proposed architectures to host applications on fog nodes and their proposed techniques on how these applications could behave to better serve their purpose. We conclude the chapter with commercial solutions that provides/employs this new computational model.

### 2.1 A New Computing Paradigm

Technology became one of the most important factors of development in our current society. It keeps improving over time, whether by devices that can communicate, such as smartphones or tablets computers; communication technologies such as 5G, long-term evolution LTE, low-power wide area networks (LPWAN), LIFI; or new concepts and paradigms such as cloud and serverless computing.

The rapid growth of connected devices is reinforced by the 2016 IHS report (IHS, 2016). It points out that in 2017 there will be 20.3 billion connected “Things” and the trend is to surpass more than 75 billion by the end of 2025. The number of connected devices by 2025 gives a share of 9.27 per person. This is supported by two major factors, cheaper electronics (Flamm, 2018) and the increased value of data collection (Giacaglia, 2019).

The evolution of lithography technology allowed more transistors per wafer, reducing the fabrication costs (Flamm, 2018), resulting in cheaper electronics. Another side effect is that the

computation power keeps doubling every eighteen months (Moore, 1965) and storage follows an exponential behavior (Klein, 2008). Cheaper and more capable devices are the driving force for the quick expansion of the connected devices (Enescu, 2014). In contrast, bandwidth does not reflect the same growth to accommodate the increasing number of connected devices' needs. Wide area communication grows 50% in its capacity, every two years (Nielsen, 2018).

On the data side, it is projected by 2021 that data centers will have 1.3 Zettabytes of stored information, 4.6-fold growth comparing to 2016, while capacity installed will reach 2.6 ZB. The devices will store more than 4.5 times what will be stored in the cloud, 5.9 ZB, and the IoT devices will be creating 847 ZB of data (Cisco, 2016). To gain a better sense of these magnitudes, if the reader has 1 Exabyte of data and wishes to move to the cloud with an Internet connection of 10MB/s, the reader will have to wait 26 years. From the numbers above, comparing the total data center capacity with the sum of the total amount of data created/stored by devices and IoT devices, we will end with a deficit of 850 ZB. We can therefore conclude that the cloud model will not be enough to cope with the demand.

In the automobile industry, we already have one company that understands the cloud model problem and already has deployed the solution. The automobile manufacturer Tesla is trying to solve autonomous driving and the current fleet is the solution. That is, they are using deep learning (Kocić, et al., 2019) to build self-driving models, those models require large, diverse and real data samples. In order to collect samples, Tesla uses the fleet to obtain the raw data. Tesla Autopilot system has 8 cameras along with other sensors, but the car does not upstream all the data that is being generated, despite data that being required by the data scientists. An example given by the Andrej Karpathy, senior director of AI at Tesla, in which he explains how they solve the problem of detecting bicycles attached to cars. Initially, when a bicycle is attached to a car, the model considers that there are two distinct objects, however, for the driving purpose, it must be considered as one moving part. So, they enquire the fleet to send images that contains that pattern, re-train the model by annotating the pattern as a single car and that particular problem is solved (Tesla, 2019). In other words, Tesla has millions of distributed datacenters where they offload computation from the "main datacenter" to each element on the fleet. The fleet generates real-time answers and upstreams the results. Karpathy also points out that cloud-centric model was not a feasible approach.

Tesla's computing model resembles fog computing. The general idea is to push the execution of the tasks closer to where the data is generated - it uses a technique called offloading. It

consists of the fact that the tasks are outsourced and the involved entities work in a tandem way to achieve the ultimate result of the application. It can occur in IoT nodes, sensors, edge devices, or fog nodes, depending on factors like application requirements, load balancing, energy management, latency, management and so on, which are evaluated to when what, where and how this technique can improve application end goals (Aazam, et al., 2018).

Fog Computing extends the Cloud Computing paradigm to the edge of the network. In (Bonomi, et al., 2011), the authors state that the pay-as-you-go Cloud model (client is charged based on usage) is here to stay, reinforcing the fact of the economics of scale (OPEX) Cloud oriented applications cannot be beaten due the strategic localization of the mega data centers. On the other hand, they emphasized the shift of the type of endpoints. Those that we use today (tablets, smart phones, laptops, etc.), to ones that will be used in future (smart grids, industrial automation, precision agriculture, etc.), named disruptive IoT.

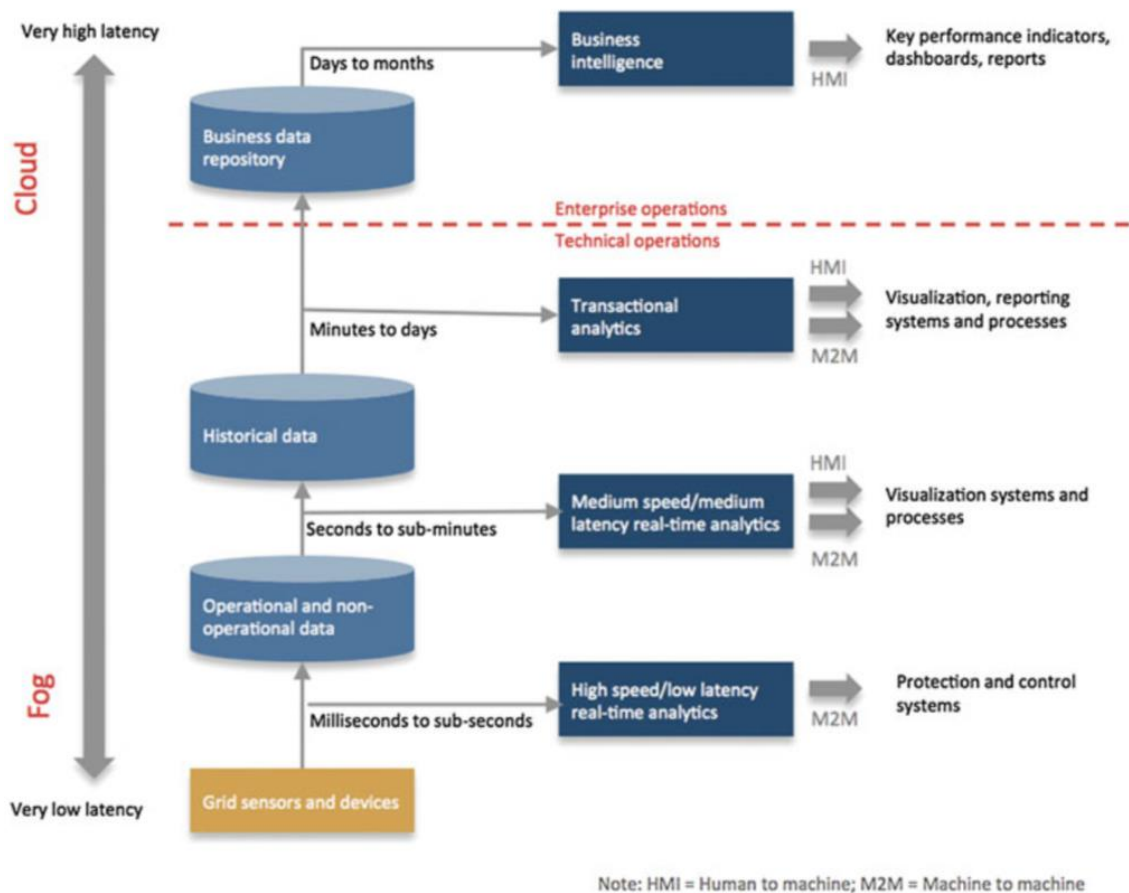


Figure 1 Fog-Cloud interplay (Bonomi, et al., 2014)

To illustrate the characteristics of this paradigm, the authors have analyzed two use cases in terms of requirements: i) a smart traffic light system, and ii) a wind farm. From those cases, they have revealed that both the systems required low/predictable latency, consistency, fog-cloud interplay, multi-agencies orchestration, and geo-distribution. The conclusion led to the heterogeneity that the paradigm must have, being an architecture of a wide range of verticals rather than being a point solution for each vertical, thus, supporting operations that run 24x7 every day of the year.

Cooperation between fog and cloud in various uses cases is crucial, and the processed data, can be narrowed in space and time at the edge and wide at the cloud, Fig. 1. The authors mentioned the Smart Grid as an example to explain the scope of the data following different time scales and how those scales dictate the type of actions and the type of communication used within those scales.

It is worth mentioning that they argued that the geo-distribution is as a new dimension of big data, not by the size or the rate of data generated by the individual sensor, but rather by the distributed nature of the sensors and actuators, where actions such as intelligent autonomous tasks must occur near where the data is generated.

Edge nodes represents nodes that are at or closer from the data is generated and pushing the computation over the edge nodes imposes new challenges. For start, in what circumstances offloading tasks make sense, in (Aazam, et al., 2018) the authors to answer this question and proposed some criteria that the implementers can use to drive their decisions:

1. Excessive computation or resource constraints;
2. To meet latency requirements;
3. Load balancing;
4. Permanent or long-term storage;
5. Data management and organization;
6. Privacy and security;
7. Accessibility.

Further in the chain of challenges, comes where and how this computation is taking place. The literature is extensive and various middlewares were proposed. They were presented with different characteristics (nodes targets, scalability, distances) to accommodate different application characteristics.

## 2.2 Fog Computing

The primary reason for this new concept is the volume of information that is generated from the connected devices (Bonomi, et al., 2014). Wide area communications, as already mentioned, grows “only” 50% in capacity every two years, consequently, not all of the data can be pushed to the cloud.

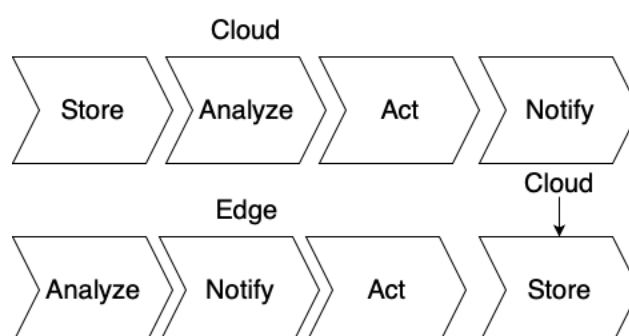


Figure 2 Paradigm shift

Fog computing is complementary to, and extension of, the traditional cloud base model. Thus, it represents a solution at the infrastructure layer to be able to deal with the amount of generated data. In Fig. 2, it is represented the shift of how the data flow will change (Enescu, 2014). Today’s the data is stored and analyzed at the cloud, and then, action is synthesized, and the edge node is notified. In the fog computing paradigm, only smart metadata is stored at cloud, and the intelligence is kept at the edge node.

### 2.2.1 Fog Definition

OpenFog consortium created at 19 of November 2015 by the IoT leaders (ARM, Intel, Dell, Cisco, Microsoft, Edge Laboratory of Princeton University), define fog computing as follows: “A horizontal, system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum.” (OpenFog, 2017).



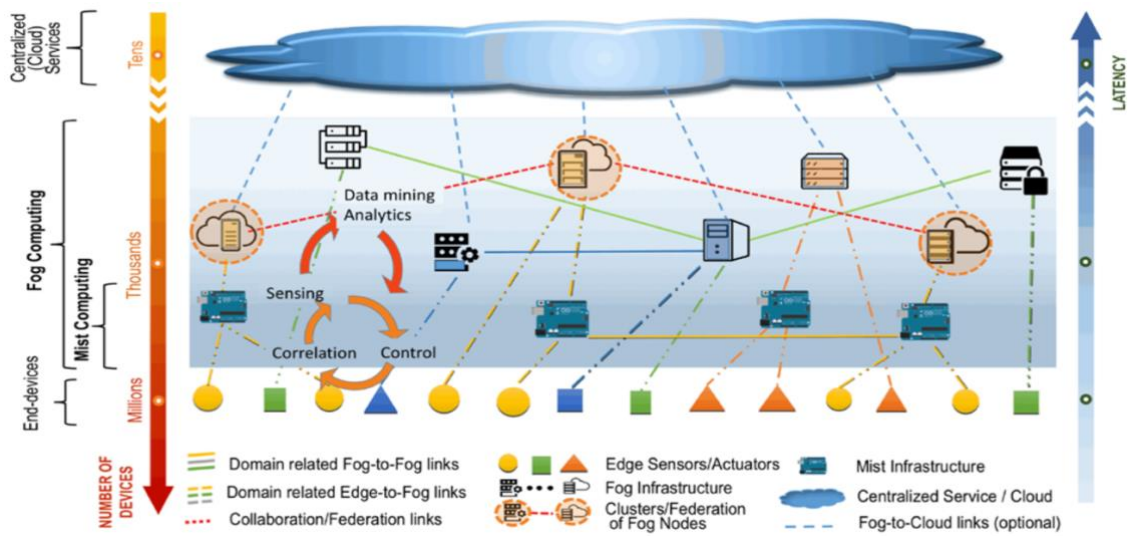


Figure 3 Architecture overview of the new computing model (Iorga, et al., 2018).

We can conclude that fog computing is a layered model, Fig. 3, for enabling ubiquitous access to a shared continuum of scalable computing resources. The fog nodes are in between the edge devices and centralized cloud services and enable easy distributed deployment of latency-aware applications and services. The nodes can be organized in clusters, either vertically to support isolation or horizontally to support federation (Iorga, et al., 2018).

### 2.2.2 Fog Computing Properties

In (Bonomi, et al., 2011), the author deduced the requirements that the new computational model has to have to accommodate the growth of the connected devices along connecting various industrial verticals. They have presented the following properties:

- Contextual location awareness, and low latency;
- Geographical distribution;
- Heterogeneity (data from different forms and factors acquired by different network protocols);
- Interoperability and federation;
- Real-Time interactions;
- Scalability and agility of federated, fog-node clusters (elastic computing, resource pooling, data-load changes, network conditions variations);
- Predominance of wireless access;
- Support for mobility.

In brief, we consider that contextual location awareness, by itself, is one very attractive property to implement autonomous and intelligence applications. A wide variety of actions/assurances depends on this fact, like smart packets routing, zero connectivity downtime, computation graph optimization, efficient M2M communication, etc.

### **2.2.3 Fog Nodes**

The fog nodes are the core component of the fog computing architecture. Each node has location awareness of its geographically and logical location within the context of the cluster. The main attributes are the following (Iorga, et al., 2018):

- Autonomous (autonomous or clustered in decision making);
- Heterogenous (different form factors, adaptative to across different environments);
- Hierarchical clustered;
- Programable (stakeholders, network operators, domain experts, equipment providers, or end users).

Fog nodes, as mentioned, can be physical components like routers, gateways, servers, edge devices, etc, or virtualized switches, virtual machines, cloudlets. Another aspect of fog node is that they provide some sort of API to the edge nodes to provide data management and communication services (Bonomi, et al., 2011).

### **2.2.4 Fog Models**

The fog architectural service models are very similar that is offered by the cloud computing model. A model is the representation of how the client uses the infrastructure. Infrastructure as a service is when the client has their own machine under a virtualized environment controlled by the infrastructure owner (AWS EC2, Google Cloud Computing). When the client only wants to manage the data and the application leaving the rest to some cloud provider, it is referred as platform as a service (Heroku). Software as a service, as the name suggest, is the software that's available via a third-party over the internet like Microsoft Office 365. The following service models can be implemented (Bonomi, et al., 2014):

- Software as a Service;
- Platform as a Service;
- Infrastructure as a Service.

In (Aazam, et al., 2018), they outline a set of applications and their tasks/data exchanges scenarios between the different computing models which can suggest which architectural model to use:

- IoTs, sensors, and devices offloading to fog:
  - Healthcare sensors offloading to fog;
- IoT/devices offloading to edge nodes:
  - Smart home devices offloading to edge nodes;
- Cloud offloading to fog:
  - CCTV video analysis;
- Distributed offloading between fog and cloud:
  - Fog real-time drone control for forest-fire and historical data provided by the cloud;
- Fog offloading to another fog and cloud to another cloud:
  - Load-balancing

In conclusion, the fog nodes undertake most of data processing at remote sites, and only forward necessary information to the cloud. As a side effect, it saves network resources and improves real-time analytics. Scalability is another aspect of moving from a centric-cloud model to a fog-distributed model by reducing network management and monitoring on current centralized architecture networks (Liang, et al., 2017).

### **2.2.5 Cloud Computing**

Cloud computing, the term is usually associated with hyper-scale datacenters. Each datacenter is strategic physical placed, crowded with homogenous computational resources (grid computing), efficient cooling systems and managed by IT infrastructure experts. From the user's perspective, the cloud hosts our data (Google Drive), enables to connect to our friends (Facebook Messenger), allows see our emails (Outlook), and so on.

In the developer's perspective, a cloud is a place where applications are deployed. The environment is a highly virtualized environment at the hardware level, forming a pool of resources shared by their users, which translates to vertical and horizontal scalability at a distance of a click on some web management platform provided by the cloud owner. The

developer pays for how much it uses, the designated pay-as-you-go model, and with this and with the scalability at his fingerprint he can provision resources according to the demand.

This computing model offers computational power, data storage, and network solutions on-demand, without having the developer to invest, setup and maintain their IT infrastructure, plus executing all other management disciplines like design and deploy disaster recovery plan(s), assuring the quality of service parameters QoS and so on (Srivastava & Khan, 2018).

The services models rehearsed by this model are the same as fog computing service models with two more: i) service as a platform, ii) platform as a platform, iii) infrastructure as a service, iv) serverless computing or function as a service. The serverless computing is a cloud computing model where the computational resources and the execution of some logic is managed by the cloud owner, like so, the developers can avoid the burden of setting up and tuning autoscaling policies or systems, making it elastic scaling, some examples of big cloud providers are: i) AWS Lambda, ii) Azure Functions, iii) Google Cloud Functions.

Wrapping up, cloud computing offers agility to the companies and developers to make their services online in a multitenant and secure environment with scalability and or elasticity to reach levels of scale appropriated by the application needs, and all of this, with the cost of how much it is used.

### **2.2.6 Edge Computing**

Edge computing is the control and the management of a standalone end-point device individually or through a set software functions in the fog domain. For example, control of a printer, security camera, traffic light, robots, machines etc., with or without a control function. Edge computing devices and entities within the domain are standalone or interconnected through proprietary networks with custom security and little interoperability (Nebbiolo Technologies, 2018).

### **2.2.7 Edge Computing vs Fog Computing vs Cloud Computing**

In Table 1, it is showed the differences between cloud and fog computing. Both have different purposes and address different use cases. However, we have outlined clear properties distinctions between them.

Table 1 Cloud and Fog computing key differences

	<i>Cloud</i>	<i>Fog</i>
<i>Architecture</i>	Centralized	Distributed
<i>Abstraction Level</i>	High	High
<i>Resource Optimization</i>	Easy	Hard
<i>Resource Management</i>	Easy	Hard
<i>Hardware</i>	Homogenous	Heterogenous
<i>Cooling Cost</i>	Higher	Lower to Medium
<i>End User Distance</i>	Higher	Lower
<i>Latency</i>	Higher	Lower to Medium
<i>Hardware Security</i>	Easy	Hard
<i>Infrastructure Ownership</i>	Well-defined	Undefined
<i>Hardware Provisioning</i>	Planned	Undefined
<i>Business Model</i>	Pay-Per-Use; Monthly Subscription;	Per computational resources
<i>Data Security</i>	Well-Secure	Well-Secure
<i>Data Analysis</i>	Long-term	Short-term
<i>Data Processing</i>	Far from source	Close from the source
<i>Connectivity</i>	WAN (TCP/IP)	Diversify (various protocols)
<i>Interoperability</i>	Web Services	Interoperability; heterogenous between resources;
<i>Customer Support</i>	Reliable	Unknown
<i>Vendor Lock-In</i>	Yes	No

The Fog and Edge computing terms are normally used interchangeably. This is because, in terms of functionality, both have the same purpose: push intelligence closer where the data is generated. However, Edge computing term was initially presented of having the control and the management of a standalone end-point device individually or through a set software function. With time, the Edge computing definition has involved, and has been redefined to include some fog properties, like interoperability and local security. Today Edge computing domain is a sub-

set of Fog computing domain (Nebbiolo Technologies, 2018). Table 2 reflects their key differences.

Table 2 Fog and Edge key differences

	<i>Fog</i>	<i>Edge</i>
<i>Across Domains</i>	Yes	No
<i>Cloud Extension</i>	Yes	No
<i>Cloud awareness</i>	Yes	No
<i>App Hosting</i>	Yes	Limited
<i>Data Service at Edge</i>	Yes	Yes
<i>Device &amp; App Management</i>	Yes	Yes
<i>Elastic Computing/ Resources Pooling</i>	Yes	No
<i>Modular Hardware</i>	Yes	No
<i>Virtualization</i>	Yes	Unknown
<i>Real-Time Control and High Availability</i>	Yes	No
<i>Security</i>	End-to-end encryption; Data protection session; Hardware Level;	Partial solution; VPN; Firewall;
<i>IoT vertical awareness</i>	Yes	No
<i>IoT vertical integration</i>	Yes	No
<i>Data Analytics</i>	Multiple devices	Single device;
<i>Anomaly Detection</i>	At edge	Undefined
<i>PLC Controller Replacement</i>	Soft PCL stack in same hardware;	Virtualizes and host soft stack;

The conclusion is that edge computing is not fog computing. They are different things, being one a subset of the other. Edge computing runs specific applications in a fixed logical location and provides direct transmission services while fog runs applications in a multi-layer architecture and decouples and meshes the hardware and software functions. Besides

computation and networking, it also addresses storage, control and data-processing acceleration (Iorga, et al., 2018).

## **2.3 Fog Middlewares**

IT users normally have one clear idea about cloud computing of being a set of inter-connected computers at a particular site owned by some big entity. The cloud owner provides facility security and management, resolves hardware failures, and uses virtualization to abstract hardware resources. All the points that are already answered by the cloud-model, however, for fog computing they remain blurred, various authors have proposed different answers to different problems. This section has the purpose to show to the reader the different execution contexts that this new paradigm can assume.

### **2.3.1 Cloudlets**

The core concepts were presented in (Satyanarayanan, et al., 2009). This middleware offers computation and storage to mobile devices in a highly decentralized architecture and dispersed infrastructure. The arguments for this design resign primarily on the WAN latency and on the premise that that will not be improved over time due to the demand-offer deficit.

The design can be viewed as a data center in a box, with a multi-core machine(s), gigabit connectivity and a high-bandwidth wireless LAN. They were idealized to be self-managed, serving few users at a time for an ideal deployment over the local business. Only soft state is preserved due the losses or destruction of the cloudlets. The distance between one determined cloudlet and the mobile user is ideally one-hop way through a high-speed wireless connection.

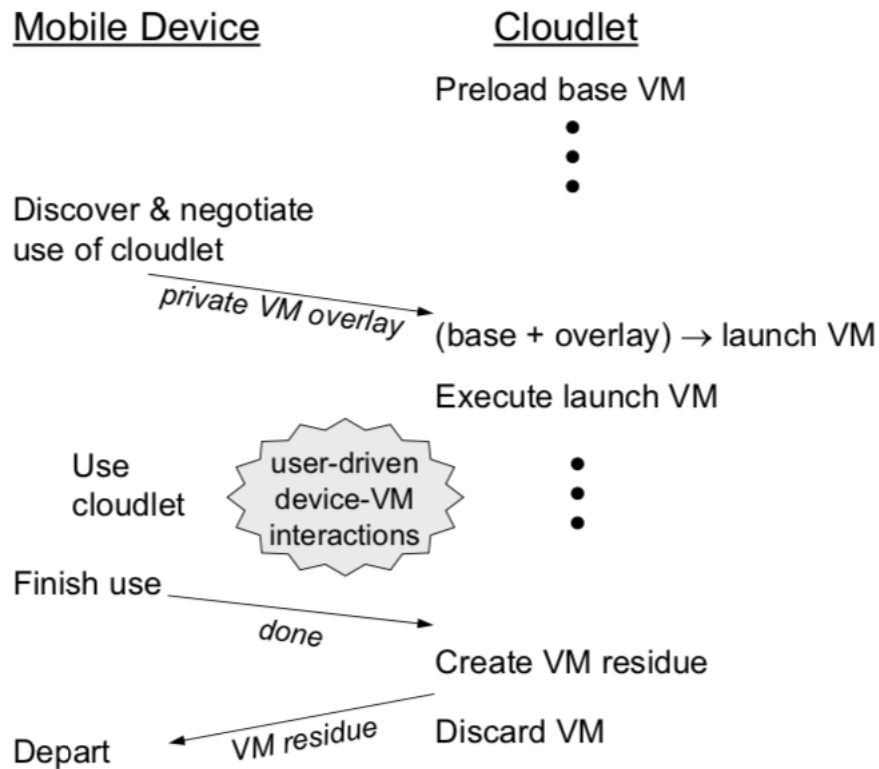


Figure 4 Cloudlet functionality interaction (Satyanarayanan, et al., 2009)

Considering the cloudlet goals, the authors have developed the proof of concept based on the method: dynamic VM synthesis. It consists of machines running a GNU/Linux operating system with VirtualBox installed and a specially developed program, Kimberley. The program launches a baseVM and when is booted, it executes a “launch-script” at the guest OS. The result is a VM that the mobile device can use. The command “resume-script” launches the client application requested by the user. From this flow, Fig. 4, results on what they have called a launchVM. Kimberley at this point computes the differences between the baseVM and the launchVM at memory and disk-level to create an overlay VM. This overlay is kept on the client-side compress and encrypted. The overlay is used at the begin of the re-incident process so that the Kimberley can synthesize the launchVM.

In conclusion, cloudlets have characteristics of targeting mobile nodes within one hop way, presenting medium to high scalability. The authors pointed that out this new architecture can lead to new types of applications which require lower latency, higher bandwidth, offline availability, and cost-effectiveness.



### 2.3.2 Mobile Edge Computing

This middleware was proposed, in 2014, by the European Telecommunications Standard Institute. Mobile-edge Computing (MEC) provides IT and cloud-computing capabilities within the Radio Access Network (Patel, et al., 2014). The MEC is characterized by three types of components: i) mobiles/IoT devices, ii) edge cloud, and iii) public cloud.

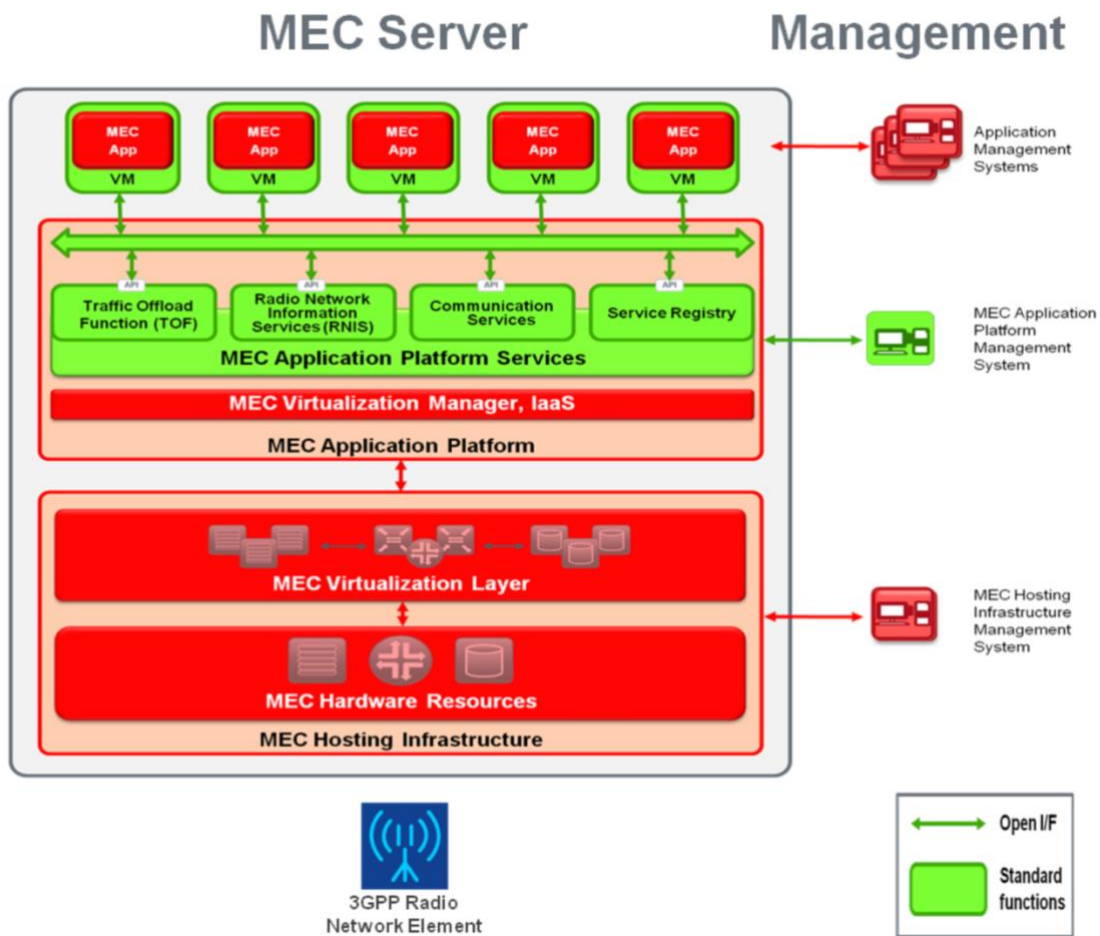


Figure 5 MEC server platform overview (Patel, et al., 2014)

The edge cloud is deployed either at LTE macro base stations or at 3G Radio Network Controller. The edge cloud can host multiple MEC applications, Fig. 5. Applications can be developed with features available by the edge cloud architecture, such as active device location tracking or RAN-aware content optimization. As a result, it improves quality-of-experience and user-satisfaction covering distances of one-hop way with high scalability properties.

### 2.3.3 Micro Data Centres

This datacenter inherits many of the properties that traditional datacenter has, such as a cooling system, fire protection, and security, however on a small scale. They are presented as all-terrain data centers, highly portable, modular and containerized, having less than 4 servers per rack, for distances of one or two hops away from the origin (Huh, 2015).

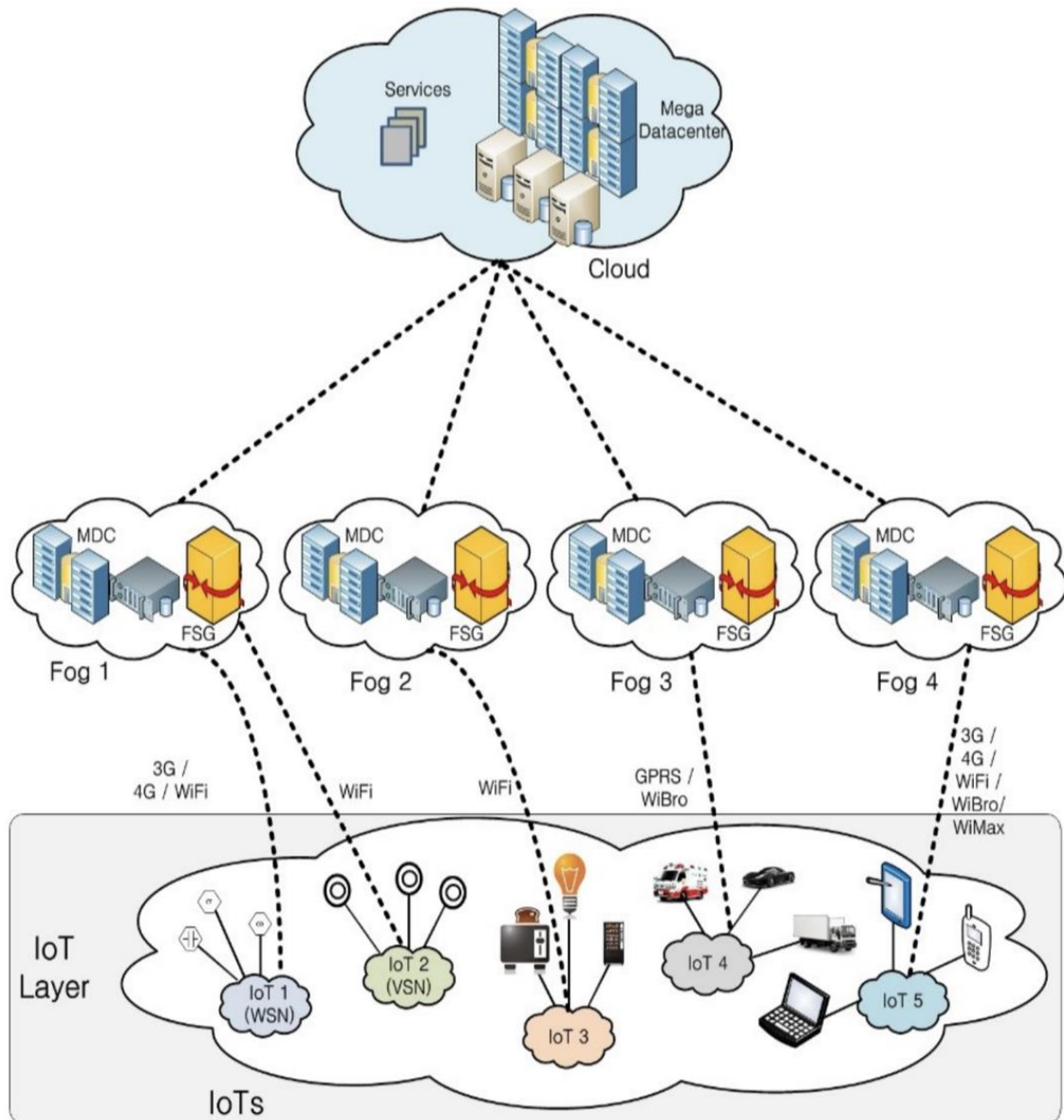


Figure 6 Fog MDC Overview (Huh, 2015)

In this architecture each fog cloud has its fog-smart gateway, Fig. 6, it does smart routing by considering applications and nodes constrains. In terms of interaction, the fog has a

set of services that can be requested by the clients (sensors, IoT, cloud). The fog-cloud acts as a market, adjusting the price and resource allocation based on client historical usage.

### 2.3.4 Nano Data Centers

Nano data centers (NaDa or nDCs) were proposed to provide energy efficiency, service-proximity and self-scalability over traditional data centers (Valancius, et al., 2009). The idea proposed by the authors consists of the creation of a P2P network (Bawa, et al., 2003), based on tiny servers located at the edge that offers storage and bandwidth. They suggest that the servers can be the ADSL gateways, the same device that connects our homes with the respective Internet Service Provider. The authors demonstrated that these gateways are 85% of the time on, Fig. 7, and in terms of energy per bit, streaming video from a Thomson Triple Play TG787v model comes at cost of  $1W/10Mbps = 100$  joules/Gb.

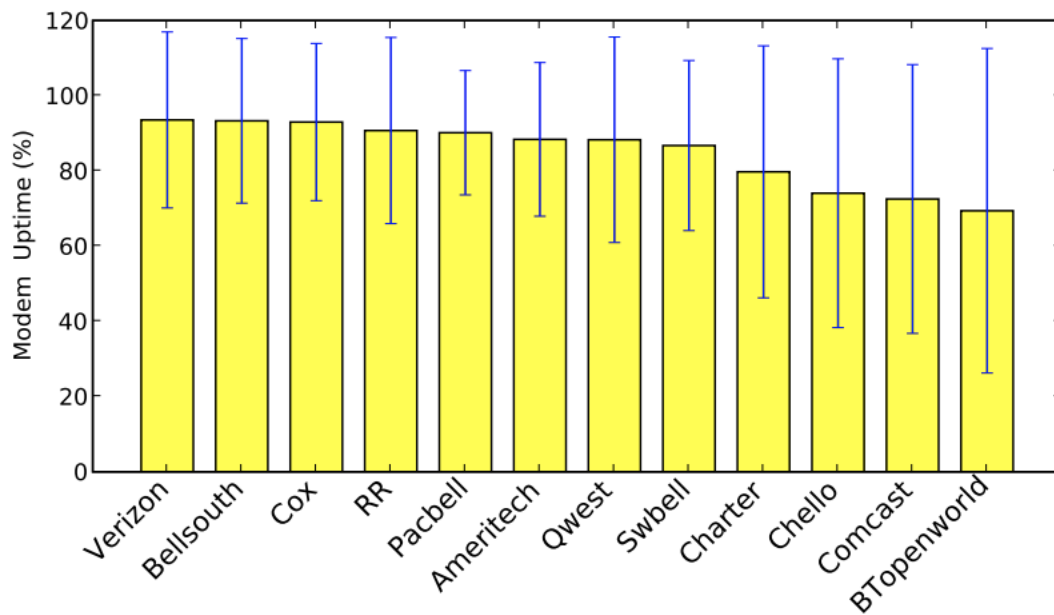


Figure 7 Modem Uptime per ISP (Valancius, et al., 2009)

The ISP in the NaDa assumes the role of a P2P tracker, which monitors the availability and content possession of the gateways. Also acts as a point of information, by matching content requests with content holders. The download is up to the gateway do execute. Another tracker responsibility is to keep gateways with the latest content, this process is idealized to take place

when network utilization is normally low. The last entity missing is the content provider. This has the role of pre-loading the gateways with content that later can be re-distributed. Theoretical energy bill cost indicates that NaDa can represent cuts of 62% when compared with traditional content delivery networks.

### 2.3.5 Delay Tolerant Networks

Delay/Disruption tolerant networks (DTN) are based on Mobile ad-hoc networks (Kurkowski, et al., 2005), in which the nodes, cooperatively form a network without infrastructure. They try to tackle long delays and communication interruptions prevalent in long distances communications. They enable connections end-to-end between devices, working on a stop-carry and forward mode. The intermediate nodes hold the data until they find a suitable node in a destination path.

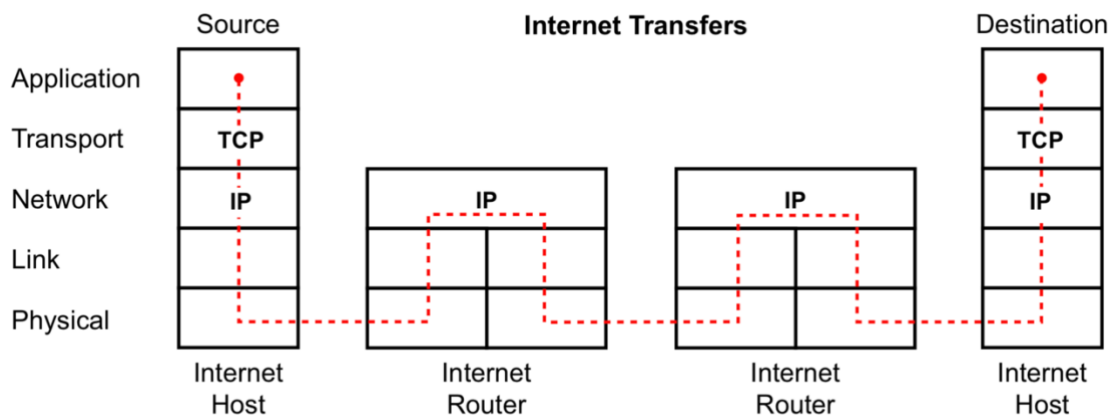


Figure 8 TCP/IP routing (Warthman, 2015)

In contrast, today's Internet is based on packet-switching, Fig. 8. Packet are pieces of data independent from each other. That is, in terms of routing, each packet can take different paths from some source to one particular destination. Routers switch those packets until reaches its destination. Consequently, packets can arrive out-of-order, but the destination's transport mechanism can reassemble them in the correct order. The usability of the Internet depends on the following properties (Warthman, 2015):

- Continuous, Bidirectional End-To-End Path;
- Short Round Trips;
- Symmetric Date-Rates;

- Low Error Rates.

On the other hand, DTNs were developed with the following network proprieties in mind:

- Intermittent Connectivity;
- Long or Variable Delay;
- Asymmetric Data Rates;
- High Error Rates.

This protocol suite adds a new layer between the Application and Transport layer, named as Bundle. This new layer stores and forwards bundle fragments between lower layer protocol, Fig. 10. Other than TCP/IP protocol stacks are suitable to coexist.

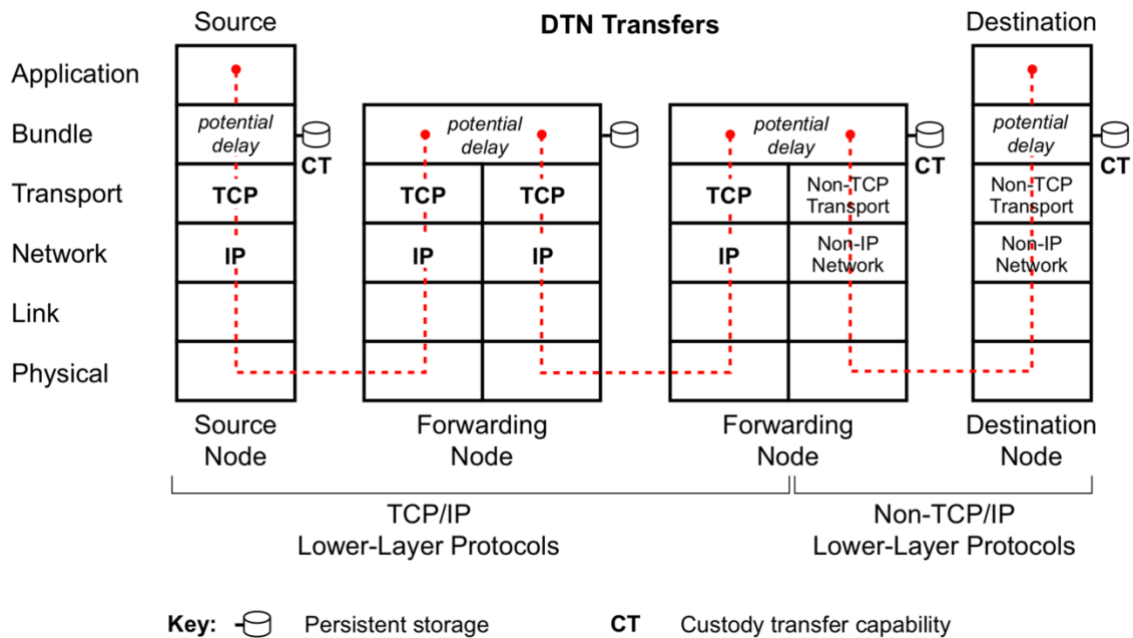


Figure 9 DNT routing (Warthman, 2015).

In brief, the protocol was being developed by the NASA Advanced Exploration System team as an open international standard (Dunbar, 2018). International Space Station already received one implementation and other 6 NASA space missions are working with it. This middleware is not infrastructure, but rather a tool to resolve one particular problem: a reliable way to transport data from a source to a destination.

### 2.3.6 Femto Clouds

In (Habak, et al., 2015), the authors proposed this middleware to harvest computational or storage capabilities of surrounding mobile devices. They state that most of the mobile devices

are under-utilized and situations like passengers with mobile devices riding a public bus or students in a classroom or a group of people on a coffee shop can provide cloud services at the edge. Femtoclouds are characterized by having distances of zero hops with the goal of task offloading with low scalability.

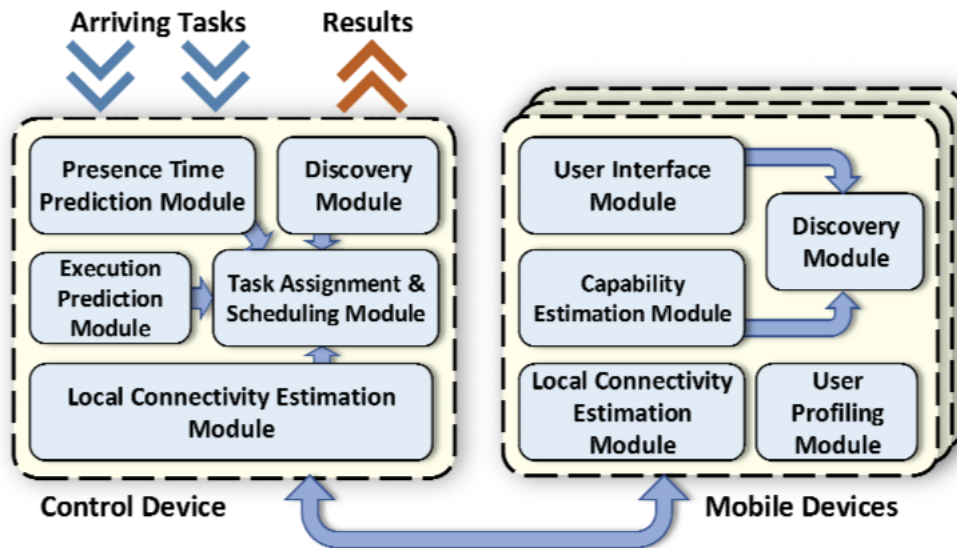


Figure 10 Femtocloud architecture (Habak, et al., 2015).

The proposed architecture is characterized by having one control device and multiple mobile devices, Fig. 10. The general idea is that mobile owners have a pre-installed a piece of software that computes the available resources to share, thus, accepts tasks to execute. The control device receives schedules, collects and responds to tasks requests. The modules and their responsibilities are:

- User Interface Module:
  - Resource sharing policies;
  - User sharing preferences;
- Capability Estimation Module:
  - Computational Capabilities;
- User Profiling Module:
  - Tracking user preferences and behavior;
  - Communicate generate profile with control device;
- Execution Prediction Model:

- Task load execution across the cloud;
- Presence Time Prediction Module:
  - Collect environment data;
  - Generic user profile creation;
  - Estimate user presence time;
- Task assignment and scheduling Module:
  - Tasks assignment;
- Local Connectivity Module:
  - Bandwidth estimation;
- Discovery Model:
  - Femtocloud client service discovery.

A scheduler, responsible for tasks assignment, was also defined in their work. Scheduling is characterized by being a NP-Complete problem, so a greedy approach was taken to tackle this issue. The scheduler is based on three ideas:

1. Prioritize tasks with higher computation requirements per unit data transfer.
2. Preferential device offloading according with their profile.
3. Dispatch as many tasks as it possible until the results gathering heuristics emits an event.

The authors conducted tests using Femtocloud architecture using 3 mobile devices and 1 tablet. The applications tested were: i) chess game; ii) video game; iii) object recognition in a video feed and iv) compute-intensive task. They measure three metrics, Fig. 11: i) computational throughput, ii) resource utilization and iii) network utilization.

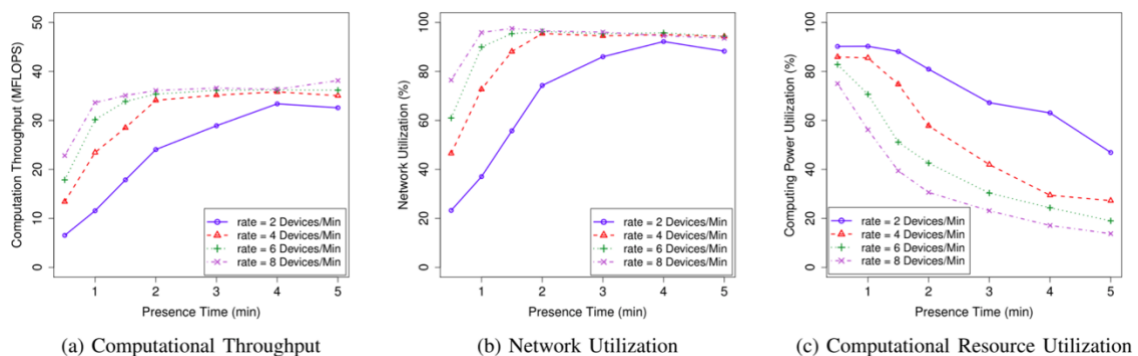


Figure 11 Femtocloud numerical results (Habak, et al., 2015).

The conclusion was that computational throughput increases in the presence of time or the user rate arrival, however, there is saturation for large numbers due to network utilization, consequently limiting task assignment rate.

### 2.3.7 Fog middleware comparison

Wrapping up, we have explored a variety of middlewares. Each possesses different characteristics and targets different computational needs.

Table 3 Fog middleware's resume

	<i>Workloads</i>	<i>Scalability</i>	<i>Business Model</i>	<i>Fog Infra-structure Ownership</i>
<i>Cloudlets</i>	Heterogenous	Medium	Execution Time;	Local Business; private households; ISP's
<i>Mobile Edge Computing</i>	Heterogenous	High	Applications Hosting; MEC services subscription;	Telcom Providers
<i>Micro Datacenters</i>	Heterogenous	Medium	Fog service Subscription	Undefined
<i>Nano Datacenters</i>	Restrict	Low;	Video Content Providers, e.g. YouTube, Netflix;	IPS's & Private Householders
<i>Delay Tolerant Networks</i>	Restrict;	low	Not Defined;	Undefined
<i>Femtocloud</i>	Restrict	Low	Task execution	Regular Users

Table 3 presents a summary from different perspectives. The criteria on categorization of workload and scalability is presented in Table 4. The workload assumes that the user is within fog-service.



Table 4 Workload & Scalability middleware's criteria

<i>Workload</i>	<i>Scalability</i>
Does a client, with no time constraints, have the possibility of executing one plus one program; How stable is the infra-structure;	Adding resources directly translates in better performance;

## 2.4 Fog Computing Techniques

In recent years, many techniques were proposed for task offloading. The authors try to answer the “how” question, envisioning different factors and application goals. The introduction of fog computing between edge and cloud adds one level of indirection. For developers using the cloud model, the questions on how to execute work are simplified, because, only one place can be addressed. The same goes for edge computing. The resource optimization is an important and difficult question to answer when fog computing comes available to public usage.

In (Zhao, et al., 2016), they state that smartphones, due to having size restrictions, reduced computation and battery constraints, the execution of complex and latency-sensitive tasks, can be offloaded to a fog cloud. In this context, they have designed an energy consumption oriented offloading algorithm for fog computing.

The algorithm computes the energy required to execute instructions in both environments: fog  $F$  and cloud  $C$ . The amount of energy serves as a threshold to forward the computation to the appropriate computing environment. In respect to fog, the energy is expressed as  $E_f(P_{tr}) = E_{f,1} + E_{f,2}$ , where  $E_{f,1}$  is the idle energy consumption when a computation is taking place, and  $E_{f,2}$  is the energy consumption for the device transfer the input bits. For the cloud is  $E_c(P_{tr}) = E_{c,1} + E_{c,2} + E_{c,3}$ , where  $E_{c,1}$  is the amount of energy for transmitting data,  $E_{c,2}$  is the idle energy consumption of the mobile device when the fog server transmits the input bits to the cloud, and  $E_{c,3}$  is the idle energy consumption during the cloud server executes the instructions.

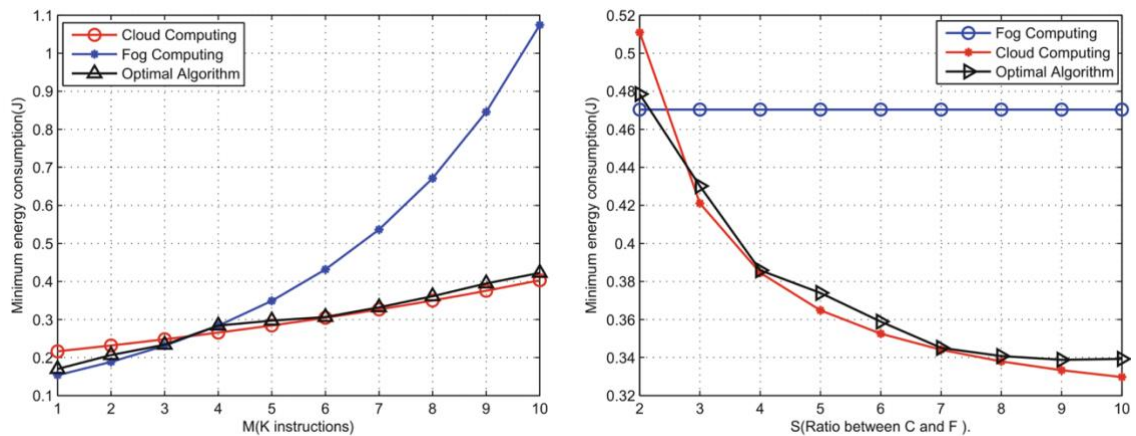


Figure 12 Numerical results of energy consumption in Zhao's paper (Zhao, et al., 2016).

The results of the scheme, Fig 12, in the left graphic, shows the number of instructions that is beneficial either to execute them in fog or the cloud. The results of using this algorithm show better energy efficiency when compared to the usage of fog or cloud computing alone. The right graph shows the energy consumption versus capacity coefficient with fog and cloud computing, where we can see the increase of  $S$  does not impact fog computing. In other hand, the cloud increases linearly, reducing the execution time, impacting this way the energy consumption.

In (Liang, et al., 2017), they have proposed a network architecture to integrate fog computing with a Software Defined Network (SDN). SDN networks abstract the routing knowledge from the physical network devices to a centralized programmable controller, consisting into 3 logic layers: i) control layer, ii) application layer and iii) infra-structure layer (Astuto, et al., 2014).

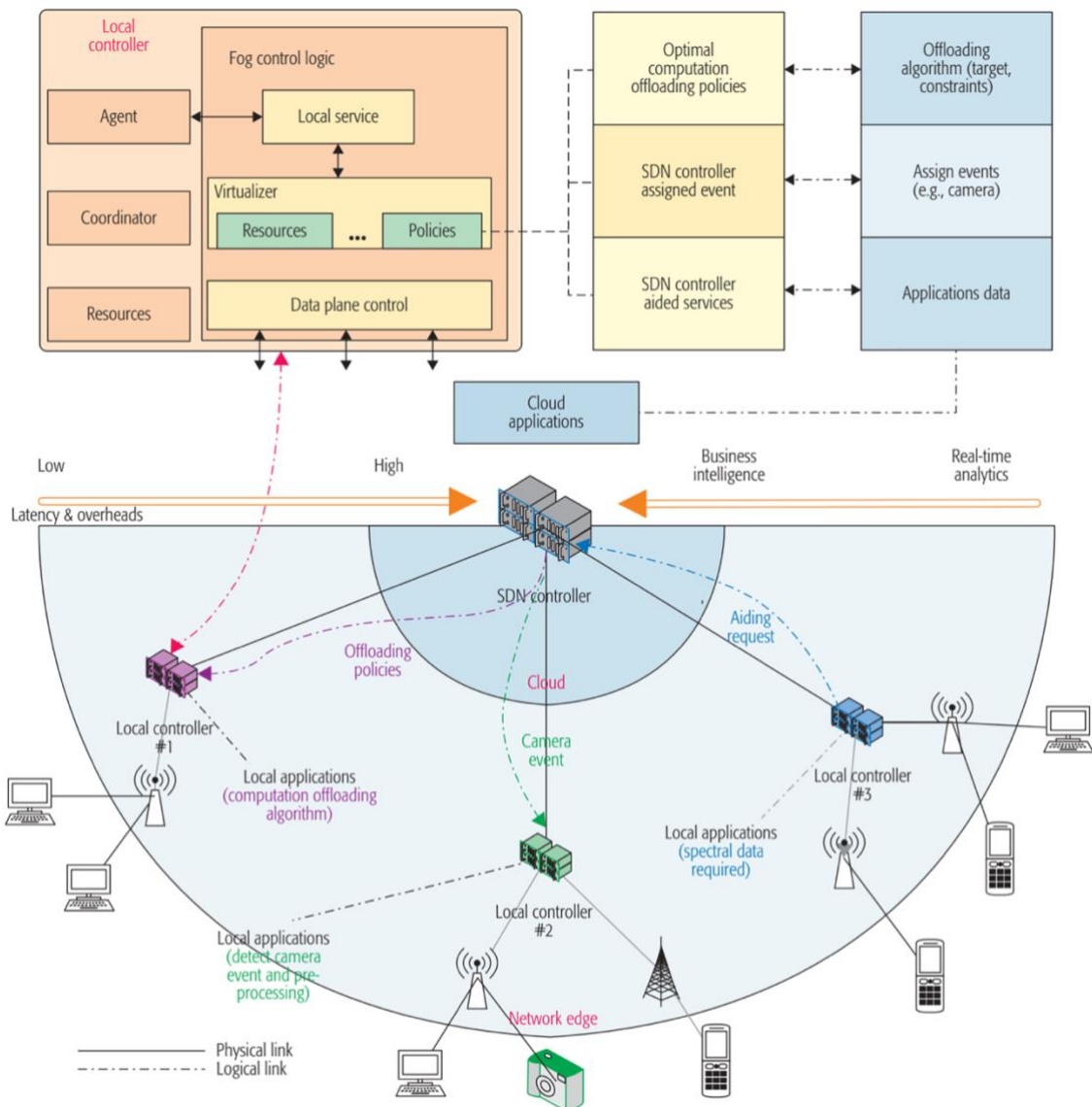


Figure 13 SDN-Fog architecture overview (Liang, et al., 2017).

The integration between the fog and the SDN is made by a hybrid control model with two hierarchical control models, Fig. 13. The local controller is responsible to provide local services or forward the data to the SDN main controller. The offloading is based on application policies and requirements like: i) latency, ii) energy consumption, iii) overhead, and iv) wireless connectivity. In their work, they have implemented the proposed architecture in a lab environment, however, there was not any kind of evaluation.

Fog data centers have fewer resources than traditional data centers, however, both can suffer from request saturation, even over-dimensioned datacenters that are designed to cope with high demand are not free from this issue (Luan, et al., 2015). In (Fricker, et al., 2016), they have

used load balancing technique to tackle request saturation based on a blocking rate offloading of tasks. An overloaded datacenter can forward to a neighborhood datacenter with the same probability of receiving the request, minimizing request rejection at an overloaded datacenter. They have demonstrated that the analytical model can improve up to 70% in rejected requests.

Aura (Hasan, et al., 2017) is a highly localized IoT based on cloud computing paradigm that uses an incentive-based model for task offloading. The users can create an ad hoc cloud using IoT devices and other nearby computing devices. To create ad hoc clouds, the author proposed an incentive and contract mechanism to allow Aura to operate in an economically feasible manner. The contracting mechanism is based on a rating system, each Aura entity (controllers, IoT devices, Mobile Agents), have a rating point between 0 and 1, the lower the number, more reliable the node is considered. The rating system is based on node participation, more precisely, in the enhancement of Aura cloud with more computing power.

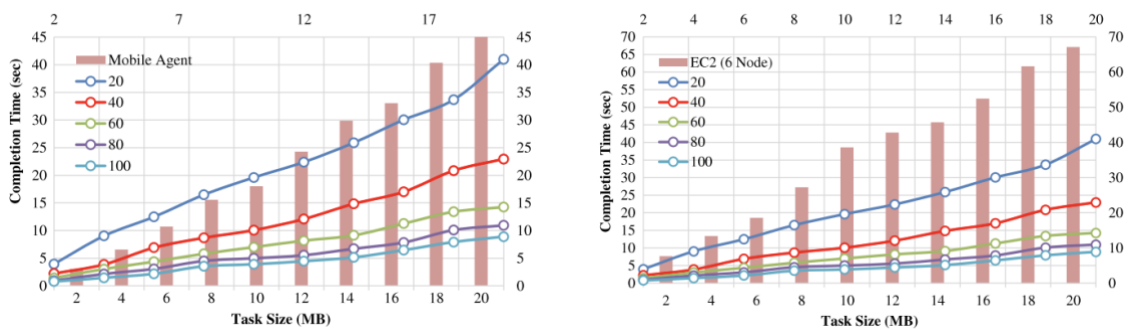


Figure 14 Aura numerical results (Hasan, et al., 2017).

To test this incentive-based architecture they executed a MapReduce Job with different sizes. Comparing the Aura and the traditional cloud (AWS EC2), they have demonstrated that Aura consumed 66% less energy when compared to the cloud counterpart, Fig. 14. Another test case was a mobile device tested against the Aura cloud with a different number of nodes (20, 40, 60, 80, 100) executing the already mentioned MapReduce jobs. The overall improvement was up to 40%-time reduction.

In (Pu, et al., 2016) they proposed a novel mobile task offloading framework named device-to-device (D2D) Fogging. D2D has the goal of to be energy efficient at task execution and with that, the authors also proposed an algorithm based on Lyapunov optimization for the minimization of the time-average energy consumption for all the users, taking into account that the system that can be over-exploited from the users. To maintain long-time contributors to the D2D, the

network is based on incentive model and employs the tit-for-tat offloading mechanism. This is when a device that offloads tasks to another node, it will have a debt to these devices. With this, it has to pay by contributing with resources for the same device. Network operators control contributions and all evolving intelligence of the offloading in the fog.

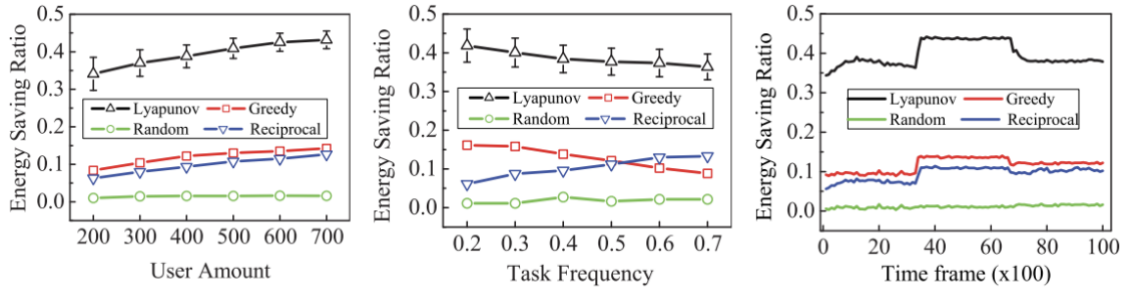


Figure 15 Lyapunov based algorithm numerical results (Pu, et al., 2016).

This algorithm was evaluated against 3 other schemes, as illustrated in Fig 15, and they are the following: i) greedy, ii) reciprocal and iii) random. The greedy at the base stations, it makes one list of all task owner-works in sorted order, and then, picks those pairs greedily. The reciprocal scheme considers users that are task owner, and in the act of exchange tasks to another task owner picks those when the overall performance is increased. The random schema creates a list of tasks owners at the base station, in random order, and chooses users randomly.

The analysis of this offloading scheme has shown savings on energy consumption of 25% with different user amount. Comparing the task frequency, the proposed scheme can save up to 30%, 23% and 18% energy over random, greedy and reciprocal schemas. The algorithm also has a 20% performance gain on time-variant conditions.

The decision making of when and where to offload resource-intensive tasks, without prior knowledge of the offloading system or a running profiler on the backend system imposes a major concern. In (Meurisch, et al., 2017), the authors look into this issue and proposed a way for the edge device to get awareness about unknown services. The methodology presented consist of offloading micro tasks to assess the network and backend capabilities. Then, with the results, the system estimates the cost and time completion for large offloading tasks. The analysis of this method has demonstrated that the system can achieve the accuracy up to 85% when to offloading larger tasks. The algorithm achieves 85% of accuracy with two micro-tasks in a range of few milliseconds.

In (Zhang, et al., 2016), the authors proposed an energy-efficient computation offloading (EECO) mechanisms for MEC in 5G heterogeneous networks. The mechanism takes into account the energy consumption that is spent during the transmission and at the execution phase. They state that during the data transmission, the energy consumption is influenced by the different states that the wireless channel can be and by the different task sizes. The users that share the same radio resource can also suffer severe interferences, consequently, transmission rates and energy efficiency are affected as well. They have formulated an optimization problem that tries to minimize energy consumption while preserving latency constraints. In their work, each task is assumed to be indivisible and when comes the offloading moment. The node computes the energy consumption either if the task was executed locally or remotely, then the node acts upon the best result. The result of their approach led to 18% of less energy required when compared with the no offloading schema.

In (Craciunescu, et al., 2015), they have discussed the different implementations of e-health applications and the importance of fog computing to accommodate them in a reliable and useful way. They have primarily addressed the problem of latency between cloud and fog. In the experiment, an e-health laboratory, data was collected from various sensors, such as pulse rate and oxygen level, and then forwarded to the fog cloud. The fog is responsible for analyzing real-time life-dependent data, which can trigger different actions, like notify caregivers with localization of a fallen patient. The fog has also fog-cloud interplay for storage patient history and retrieves it when was needed as well. The testing results showed that for the same task when offloaded to the cloud, the latency, increases by 2s to 4s compared to the fog computing middleware approach.

Another incentive-driven computation offloading model was proposed by (Liu, et al., 2017). They outline the importance of getting participants to join and let other participants consume their resources. Without it, the feasibility of the network could be compromised in terms of the existence itself. With this, they have analyzed two main issues: i) determine if the cloud server forwards or not the computation into the fog, and ii) how big is the reward to the surrogate fog node. They have answered to those questions by formulating a non-cooperative game theory between two entities: i) cloud service operator, and ii) edge server owners. The Stackelberg game has two steps: i) the cloud specifies a payment profile, and ii) the fog node answers with the amount of its computation for offloading based on the payment.

The equilibrium of the system was systematically evaluated, and they have demonstrated that that system can achieve the Nash equilibrium. At the Nash equilibrium, the cloud service operator maximizes its utility based on the optimal strategies of the edge server owner. Each edge owner is able to maximize its utility by selecting the optimal strategy. Therefore, the optimal strategy is the equilibrium strategy for each edge server.

In conclusion, fog computing still is in its infancy. The ownership of the fog Infrastructure in some works suggest the bottom-up model (Hasan, et al., 2017), where the fog nodes are owned by IoT users. In contrast to bottom-up is the bottom-down, in other words, the cloud providers which install computational resources at the edge (Zhang, et al., 2016). It is worth to point out that there is a convergence for incentive-based schemes, suggesting a third model where both models can coexist. The ownership problem was presented by the creator of this new computing model (Bonomi, et al., 2011), and it remains an unresolved one. In this undecided environment, the main focus is to abstract the ownership and focus on major technical challenges, like energy consumption schemes, addressing delay and latency constraints and computation needs in an application-fog-cloud symbiosis.

## **2.5 Commercial Solutions**

The fog computing model is not being staved off by big companies. Many companies have already commercial solutions. Cisco leads at the hardware level, introducing fog capabilities on their network devices, while some other companies are focusing on the software side. We also have mentioned what big cloud providers have to offer, even if the current solutions have not been designed as a fog computing paradigm, yet it can be a serious component with the already existing IoT computing infrastructure.

### **2.5.1 Cisco I0x**

Cisco Systems has launched a new platform for IoT and Fog Computing, namely The Cisco I0x. This technologic combines the new computing paradigm with the Cisco Network infrastructure. Equipment's from four different lines have well-suited Fog features: i) Cisco 800 series industrial services routers; ii) Cisco industrial ethernet 4000 family switches; iii) Compute module for Cisco 1000 series-connected grid routers; iv) Cisco IR510 WPAN industrial router. The devices are built with support for LXC and/or other types of virtualization (Virtual Machine, Docker) (Cisco, 2016).

Packaging IOx fog applications involves the creation of a package descriptor file which contains the resource needs and other metadata. This file is under the project directory, and then, a specialized client interface tool “ioxclient” is used to build the application artifacts under “artifacts.tar.gz”. Finally, the CLI tool wraps up the project folder and the resultant file is able to be deployed under IOx device.

Deploy is made by the Cisco Fog Director. This is a rich web-based administration application that enables the users to monitor and collect statistics. This application also offers the possibilities of the user to install, remove, backup, debug and manage application life-cycle. Fig. 16, illustrates the different states that application can be in.

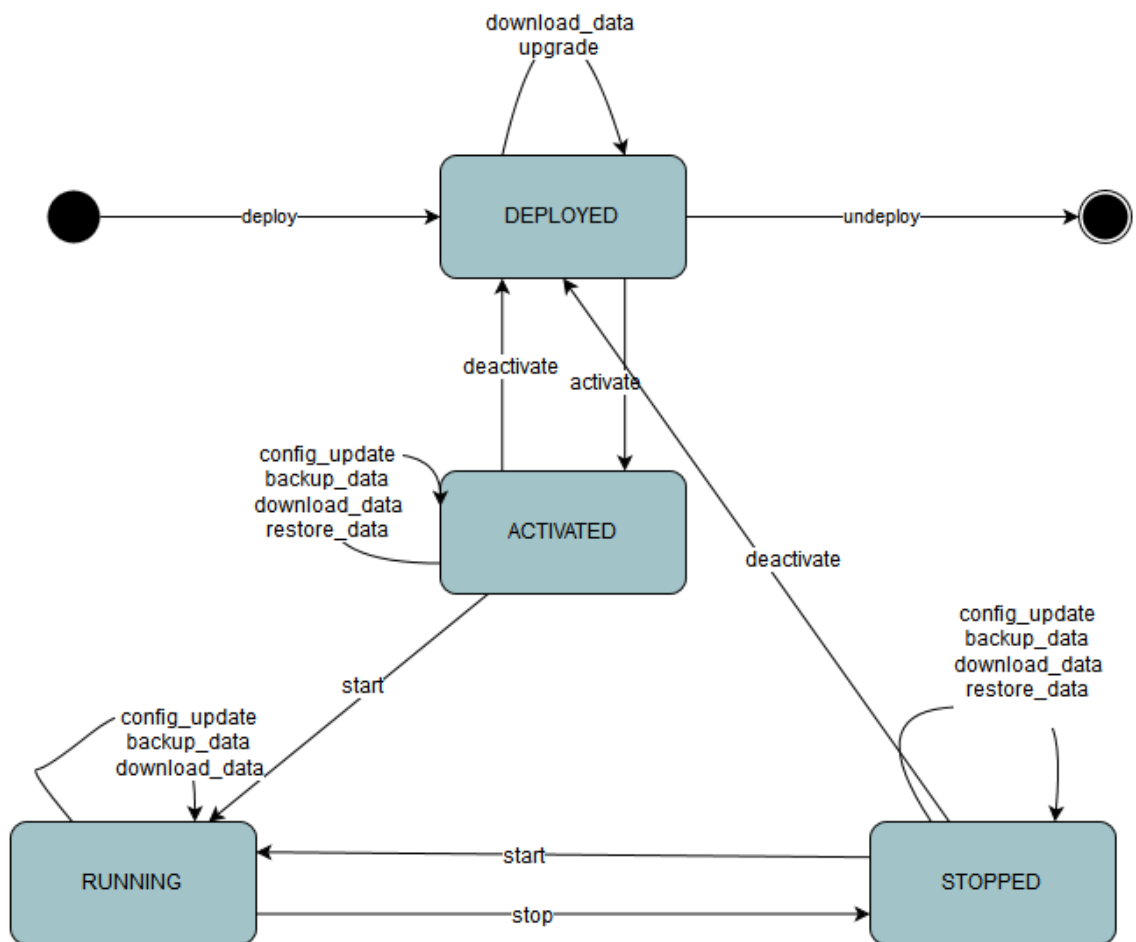


Figure 16 Cisco IOx application life-cycle state diagram (Cisco, 2016).

Cisco IOx platform is not just for hosting applications at the edge, it also offers out of shelf services that the application developers can use: i) secure storage, ii) GPS, iii) motion detection, iii) ModBus, iv) DN3P and v) service discovery. The fog application can consume these services by a well-defined REST or WebSocket API. This homogenous way of service consumption is done



by the integration layer. This layer provides a message broker model to perform remote procedure calls RPC and publish-subscribe schemes between services and northbound interface NBI. NBI acts a gateway either from IOx applications to NBI nor services to NBI, translating the communication accordingly. The services are not only provided by the vendor but can also be developed by the community.

Summing up, Cisco IOx offers an off-the-shelf platform and infrastructure to execute fog applications. The principal advantages of this technology are familiarity, in that the application developer implements and use existing tools such as Docker.

### **2.5.2 FogHorn**

The full scenario of this solution consists of a machine learning model, or other AI model, that is trained at the cloud, and then is edgified and transferred to the FogHorn stack. The stack runs at edge node(s) and performs inference over streams of data. The insight generated at the site is pushed into the cloud and automatically updates the model without human scientist interference (Guilfoyle, 2018).

The solution (Hughes, 2017) is at the software layer. The software is independent and cloud agnostic. The FogHorn's VEL Complex Event Processor lives at edge node and the main functionalities are: i) data pre and post-processing, ii) closed-loop actions control; iii) pattern recognition, and iv) fog-cloud interplay. The tool that edgifies the AI or ML model is EdgeML. This tool is capable of reducing the resources needed in the edge node up to 80%, enabling devices with limited resources to host complex models.

The reason that we have outlined this solution, resides on the fact of its capability of prescriptive maintenance capabilities (King, 2018), Fig. 17. This solution is already employed to optimize machines workloads, like production throughput, machinery lifetime, energy costs, etc. Reinforcing the idea introducing by (Evans & Annunziata, 2012), when they state that energy efficiency can only be achieved with intelligent machines and better data analytics.

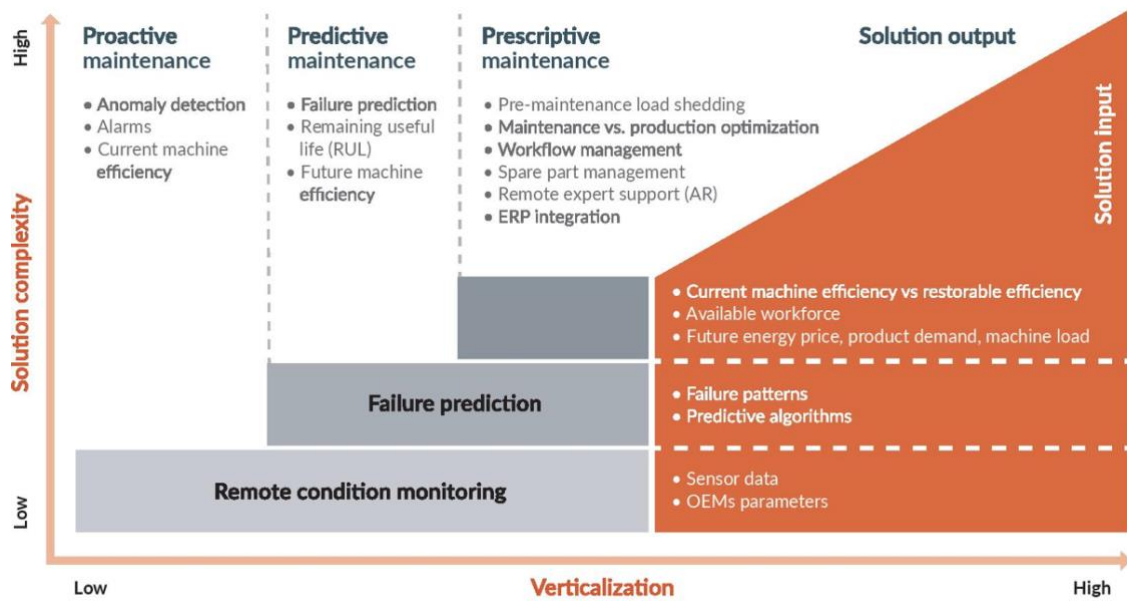


Figure 17 Maintenance Types (Hegde, 2018).

### 2.5.3 Big Cloud Providers

The solution “Lambda@edge” from Amazon has the potential of improving user experience. They leverage their content delivery network to host and run the lambdas. The lambdas are not for generic use, but instead, they are aimed to control/serve and/or static content.

In the IoT field, all major cloud player has an IoT solution(s), table 5. Bosh, AWS and Microsoft are very strong in terms of the number of the available solutions, others like Google Cloud Computing (GPC) and Alibaba Cloud offers a single and similar solution.

Table 5 Cloud providers and their IoT solutions.

	<i>Microsoft IoT</i>	<i>GPC/Alibaba</i>	<i>AWS</i>	<i>Bosh</i>
<i>IoT Solutions</i>	Azure IoT Central; Azure IoT Edge; Azure IoT Hub; Azure Digital Twins; Azure Sphere;	IoT Core;	Amazon FreeRTOS IoT Greengrass; IoT Core; IoT Device Management;	IoT Communication Suit; IoT Software Updates; IoT Analytics; IoT Hub; IoT Insights;

	Azure Time Series Insight; Azure Maps; Event Grid;		IoT Device Defender; IoT Things Graph; IoT Analytics; IoT SiteWise; IoT Events; IoT 1-click	IoT Permission; IoT Remote Manager; IoT Rollouts; IoT Things; IoT Gateway Software;
Payment Model	Diversify (monthly, per-use, per-message, per-node)	Number of messages	Diversify (monthly, per-use, per-message, per-node)	Monthly (each service)
Maintenance Type	Predictive	Predictive	Predictive	Predictive

We will not discuss each solution, per cloud provider for two reasons: market volatility and theoretical relevance. Due to being generic and given its characteristics, we will cover in more detail Microsoft’s IoT Hub solution.

## 3 Solution Description

This chapter is where the solution is described, from the design up to the implementation. It starts with the vision of the solution. Then is specified their functional requirements, followed by an in-depth technology study an concluding the implementation aspects of the proposed solutions.

### 3.1 Vision

We envision one solution able to integrate millions of on-premises devices with the existing cloud-infrastructure to assemble a network of devices capable of providing new services. The solution must represent what is considered a fog architecture, tying the cloud and the edge devices together. To support the new paradigm shift, it must produce comprehensive set of analytics to measure the benefits of the adoption.

For this vision, we have adopted serverless architecture to incorporate third party nodes that host serverless applications also cloud be referred as lambda functions, and this last term is the last one that will be used in the solution description. The lambdas will support two types of events:

1. HTTP Request Event
2. CRON HTTP Response (HTTP client)

The solution must offer an easy and intuitive interface to the user perform lambda management, such as create, edit, list and delete. When creating a new lambda, it must be specified what type of event that triggers the lambda, also, it must be possible to choose in what environment (programming language) will be coded and their respective run-time dependencies.

The client controls where the lambda is deployed. For that, a list of the registered nodes with valuable information is displayed. The client can select multiple nodes and control the number of instances that each node will host. This way, the platform has to manage the deployment to the specified nodes with the respective number of instances.

The platform must be created with a fall back mechanism in case of unexpected nodes shutdowns. Tracking nodes and their lambdas status is the first step. Another move is to re-deploy individually lambdas in case of abrupt termination. Finally, when a node restart, its previous state must be restored.

In term of stats, the client must be able to see, in a form of logs, all deployment lifecycle and individual lambda execution lifecycle. Metrics must be reflective in those logs are:

1. Memory Usage;
2. CPU Usage;
3. Bandwidth Usage;
4. Execution Times;
5. Event Trigger Request;
6. Event Trigger Response;

The platform must be built with reactive properties. Responsive so the client can obtain responses promptly. Resilient in face of failure. Elastic under different types of load and message-driven implying asynchronous technologies for the development. The code must be tested and documented.

### **3.1.1 Positioning**

#### **3.1.1.1 Business Opportunity**

Empower clients with a tool capable of interplay with the cloud and edge devices. Refactoring cloud business model by introducing edge devices as first-class citizens. The change from cloud-centric (centralized) to fog (decentralized), moves computation closer to the users with the potential of cultivating new user experiences and services.

#### **3.1.1.2 Problem Statement**

The cloud became the main vehicle to host all sort of computational needs, on top of that, its pay-per-use model makes it very attractive and irresistible to not to use it. Still, there are applications where the cloud model does not fit. Primarily, applications that latency has a major impact or applications that generate big quantities of row data are not contemplated by the current model.

#### **3.1.1.3 Alternatives and Competition**

In general way, all cloud providers, Cisco IOx (software and hardware) solutions mentioned before are part of the competition. AWS IoT Greengrass is the most direct competitor in terms of execution flow, not in functionality. AWS IoT Greengrass that lambdas can be deployed and executed on the device and communicate with other devices without cloud mediation (within LAN).

### **3.1.2 Stakeholders Description**

The solution targets three different stakeholders. The public user is the stakeholder who wants to use the platform as a software as a service. This stakeholder does not provide any computational resource although, he wants to execute logic by using the existing infrastructure, accessing to the analytics and manage their logic units. He relies on the existing infrastructure to deploy services.

The hybrid stakeholder has computational resources and makes it available on the platform for his uses and other users as well. He benefits from the analytics available in the platform, another aspect taking into account, is that he owns part of the infrastructure, so he has a higher level of confidence when running logic units in his nodes. He also is responsible for the node(s) maintenance.

The private stakeholder, like the hybrid one, he owns computational resources, however, it does not have intentions to share them, whether for legal or political reasons. He wants a private solution for private usage. Ultimately, he wants a replica of the existing solution without third parties accessing his resources.

Table 6 High level goals

High Level Goals	Priority	Problems and Concerns	Current Solutions
Integrate multiple nodes from multiple users for multipurpose uses	Hight	Scalability; Security; Complexity; Incentive to make those nodes owners to remain in the network; Nodes Maintenance;	None;
Integrate edge node with cloud nodes	Hight	Simplicity; Resource optimization;	All solutions analyzed;
Comprehensive analytics	Medium	Aggregates query plan;	All solutions analyzed;

The table 6 summarizes the high-level goals for the different stakeholders where was identified the problems and concerns to be achieved by this new platform prioritize them according. The solutions also are identified by each goal.

### 3.1.3 Product overview

The product, Fig. 18, has three independent parts, each part offers a specific set of functionalities. This division addresses different user needs and resources. In the segment side, the product has two major segments, community and proprietary.

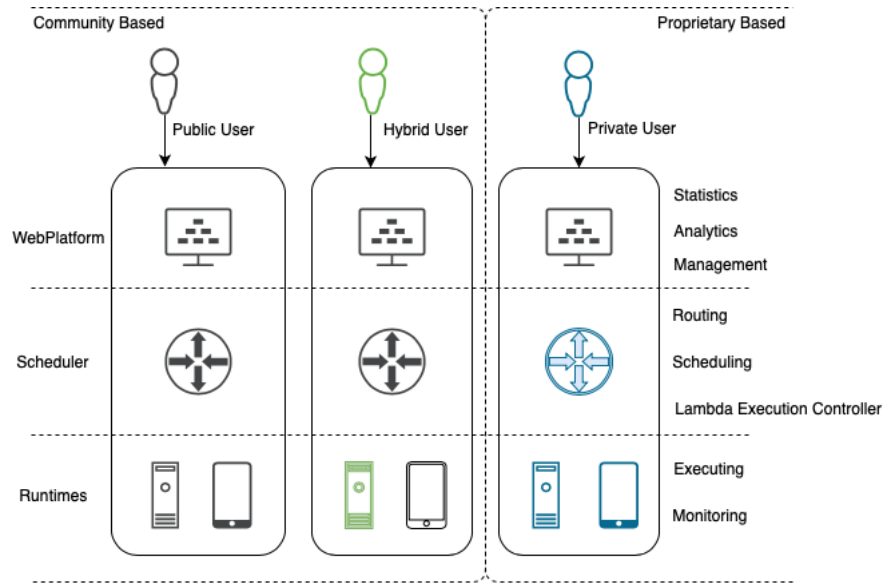


Figure 18 Product overview

The Fig. 18 also has different colors (green and blue) for different infrastructure ownership, where we can see the hybrid user in green and their infrastructure part to represent computational resources as green as well, and the same goes for the private user with blue color.

The community, where all resources are shared between platform users, and the proprietary where is owned by some person or group. The visibility of the elements (network, nodes, communication) is impacted by the segment that is in.

### 3.1.4 Value Analysis

The value analysis was based on the work (Koen, et al., 2001), where the author describes one front-end innovation process to formulate and reasoning about the product that we are proposing in this work using the new product development model (Dewulf, 2013).

#### 3.1.4.1 Opportunity Identification

Fog computing is to offer compute, storage and network services between end devices and traditional cloud computing datacenters, typically, but not exclusively located at the edge of the network (Bonomi, et al., 2014), with this, and from what we have researched, the ownership of this infrastructure is not well-defined and the convergence to incentive-based schemes,



suggests that this fog infrastructure will be the convergence of the top-down and bottom-up ownership models.

#### 3.1.4.2 Opportunity Analysis

Several factors add the need for offering computational resources under closer to the end devices. We identified the following ones: i) increasing number of smart devices, ii) latency between edge and cloud nodes, iii) data moving costs, iv) faster data insight, v) computational resources placement, vi) ownership. We already covered those factors, and those make an opportunity to build a new product that addresses those issues.

#### 3.1.4.3 Idea Genesis

The idea is simple, offering computation, storage, and network services between end devices and traditional cloud computing datacenters by creating one solution that allows the creation of applications that can use the resources in the best possible way. The application development under this solution must be based on a familiar and existence flow for the developer, like this, the learning curve for the usage and development under this solution be minimal. On the resources side, the solution must provide a simplified process for aggregating resources, favor even non-technical people that want to be their devices harvest by third-party applications.

#### 3.1.4.4 Idea Selection

Function as a service model was the selected architectural service model. This model is familiar to the developers, plus, this model enforces the pattern of single responsibility principle. We also added selective deployment to this service model, allowing the developer to deploy each developed function/lambda on the preferred physical node.

#### 3.1.4.5 Concept and Technologic Development

The idea is to create the building blocks using open source technologies and make the product public available for fomenting other developers to use and contribute to constituting a community.

## 3.2 Requirements

This section is more technical and is where is explained the functionality of the solution envisioned. This section count with the description of the domain model followed by the uses cases of each platform system ending with the supplementary specification.

### 3.2.1 Domain Model

The web platform is an interface that allows clients to create lambdas functions. Each lambda is executed when an event occurs. Events like the file system, network, sensor and timing events are used to start one lambda execution. The client defines the amount of memory, storage and CPU that the individual lambda will consume. The client codifies the handlers to those events.

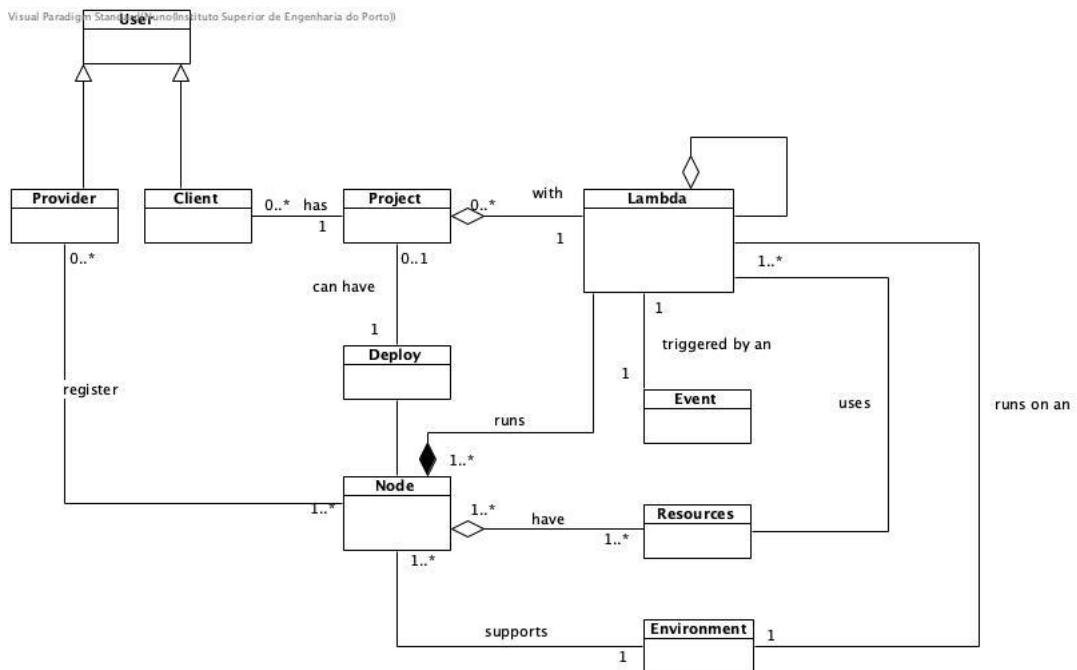


Figure 19 Web platform domain model

The client also decides the programming language that each lambda use, and for it, it can add dependencies. The previous paragraphs describes the domain model, Fig. 19, which is incorporate the main entities involved on the business of the web platform.

The platform to increase client productivity uses the popular dependencies managers available for each supported environment to search and only accept dependencies that are available on public repositories.

To help the client focus on the business logic rather on the implementation of individual lambdas, all pre-created lambdas are available to be used when a new one is being created. The

client can choose what lambda(s) will run before and after in 2 different ways: sequential and parallel.

The parallel execution can be configured to run a set of lambdas before and/or after according to the target lambda. The semantics is that this lambda event has a side effect and wants that other lambdas to be aware of the execution of this event. The client can choose one of the following execution strategies: i) fire and forget, ii) wait for the first, and iii) wait for all. The number of lambdas indicated in after or before is restricted by the total number of lambdas that the client has previously created. The platform transforms the specified lambdas into a set of that will run before and after. The parallel lambdas never interfere in the success or failure of the overall lambda execution, in other words, this lambda must always be executed.

The execution strategy of fire and forget reflects a set of lambdas that are launched, and the main lambda execution continues without waiting for their results of the launched lambdas. The wait for the first is when a client defined a set of one or more lambdas and the first result between the two or more, resumes the execution of this lambdas. The wait for all, as the name implies, waits for all results and only then, the execution is resumed.

The sequential, the semantics it that this lambda needs something from another lambda, and the nature of the dependency can be strong enough to this lambda never be called or to result in failure even if this lambda succeeded in doing their work. The order depends if it is after of before this lambda. The client can define the following execution strategies: i) continue even if the lambda fails, and ii) halt the execution. One important aspect is if the lambda halts is execution, the after parallel lambdas are never executed. The client only can be specified one lambda to run after and/or before.

The arguments are equally managed for the sequential and parallel part, the client can specify that he wants to inject the event as lambda argument and/or inject the result of the lambda into the next lambda execution.

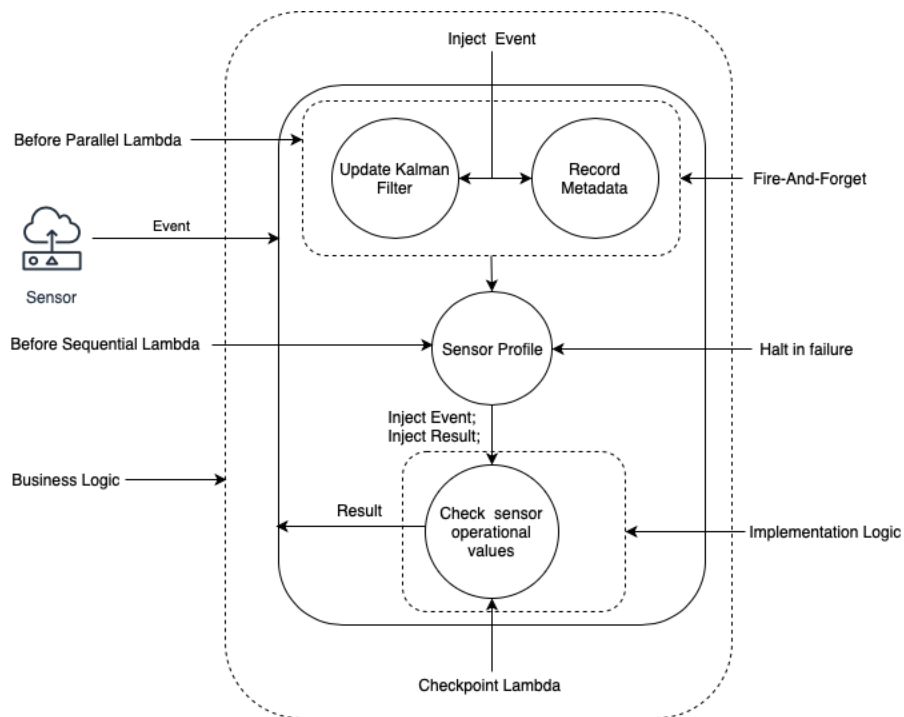


Figure 20 Example usage scenario

Consider the scenario of sensor that periodically sends events and the client wants to check if the operational values conform with their sensor profile, Fig. 20. The client creates the lambda with the logic of validating sensor profiles against the actual sensor operational readings. The client has already created one lambda that retrieves sensor profiles, so we can specify that this lambda is dependent of the result of the sensor profiler retrieval lambda, adjust the execution behavior and obtain the inputs necessary to run the logic. Meanwhile, other lambda updates some model to posterior logic and a record keeper persists the event.

The lambdas are within a domain. The domain has a set of lambdas and each lambda can contain those lambdas. The client deploys a domain as a whole and each lambda can run in a node(s), the client has the best knowledge to arrange the pairs lambda-node. The node has resources and implements a set of environments and can host multiple instances of the same lambda. The domain is designed by its name and description. The domain name is unique across the platform.

The provider contains a set of nodes, and the platform allows the registration/management of the nodes. The provider and the client both are platform users. A user is an entity that the system can identify unequivocally with some relevant information.

Lambdas run at different nodes, a separate entity is responsible to mediate the deployment and the execution of those lambdas, Fig. 21. One deployment can contain multiples lambdas, the representation of the set of actions that one deploys may need is represented as a graph. The graph is constituted by lambdas (nodes in graph theory) and their properties as edges. The scheduler executes those graphs as tasks and each task as a stats reporter that emits useful information.

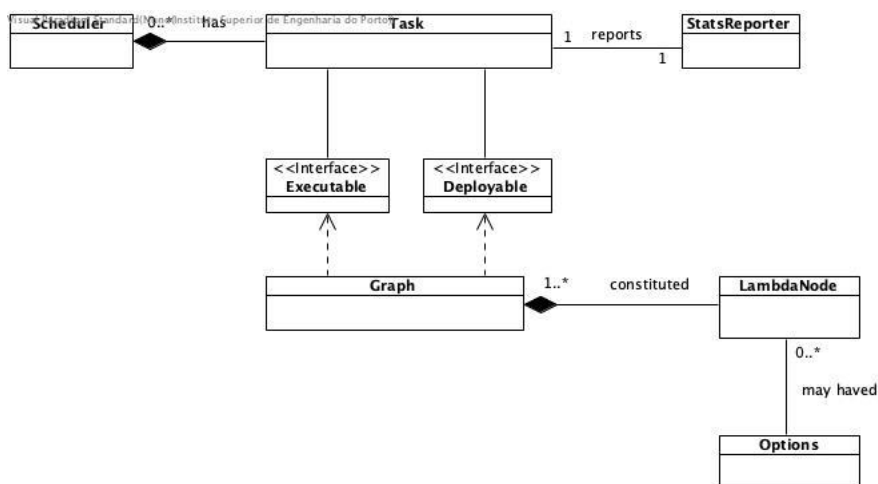


Figure 21 Scheduler domain model

When a consumer sends an event to be processed, this system uses the same graph to call the lambda, who is responsible to process and responds to the event. A lambda execution is a flow of information that is potential process by multiple lambdas, due to the ability to express complex flows within a simple call, the scheduler must remain reliable in each task execution.

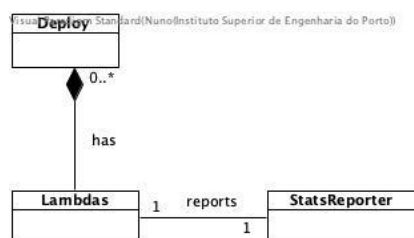


Figure 22 Runtime domain model

The part that resides on-premises is the actual worker, Fig. 22. The worker performs all the actions needed to deploy and keep running the lambdas functions. The deployment process is associate with each lambda and its state is reported back to the web platform.

### **3.2.2 Use Cases**

Were identified four different systems. I) Web-Platform; ii) Scheduler; iii) Runtime; and iv) Lambda. The Web-Platform is an interface that captures the needs of the client actor. It provides the functionality to allow the client to express and orchestrate their computational needs. In other hand, it also interfaces the resources that providers have. The Scheduler as the name implies, it schedules the task(s) to run on the provider's nodes. The Runtime is the part that is responsible for providing a common setup to all lambdas that are destiny to run on that particular node. Lastly, the Lambda is where the client's work is executed.

In term of actors, were captured seven of them. The Web-Platform interacts with three different actors, i) user, ii) provider and iii) client. The system Scheduler interacts with the deployer and consumer. The Runtime interacts with time, provider and with deployer actors. Finally, the Lambda only interacts with the consumer.

The user represents either a person who can be a provider or a client. The client represents the entity who wants to use and distribute computation using the web platform has a medium to achieve that. The provider is the entity who wants their computational resources to be used by the clients. The deployer is the entity in charge of deploying a given project. The consumer represents everybody who needs the result(s) from a given lambda. The time was also mentioned since the nodes have multiples lambdas deployed and their execution is monitored and reported to, later on, meaningful information can be displayed to the client.

### **3.2.3 Web Platform Use Cases**

In the Web-Platform were identified nineteen different use cases, Fig. 23. Is in this platform that the two main actors will interact, client and provider. It identifies them by letting register into the platform. Both have different roles, and each has a set of available actions.

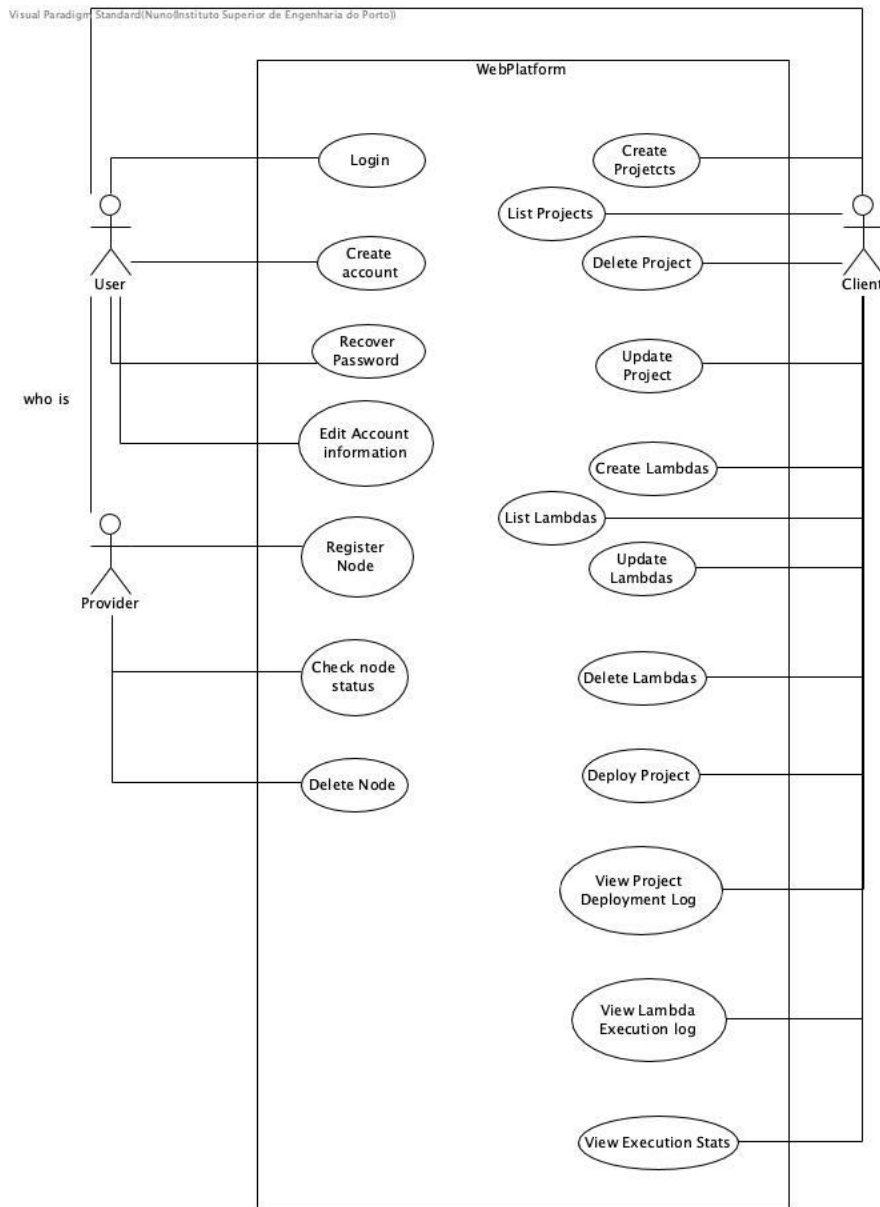


Figure 23 Web platform use cases

To the client was twelve different use cases, in counterpart, the provider has three and both have the remaining four. It is reasonable that the client at this stage has more use case because we are focusing on the engineering part of the fog architecture.

### 3.2.3.1 Create Lambda Use Case:

Primary Actor(s):

- Client

Stakeholders and Interests:

- Client: Wants a fast, easy and secure way to code services;
- Consumer: Wants a familiar form to call the service exposed by the client's lambda;

Preconditions:

- Client is identified and authenticated;
- At least one provider node registered;

Success Guarantee:

- Lambda is saved in the system;

Main success scenario:

1. Client start the process of lambda creation;
2. The platform requests lambda information (lambda name, lambda functionality, number of instances; environment, resources, node(s) to target deploy; event trigger; list of lambdas to run before and after in parallel way; lambda to run sequential before and after, lambda arguments; list of dependencies, lambda code);
3. The client inserts all information and submits;
4. The platform validates and informs the success of the operation;

Extensions:

- a\*. at any time, platform fails:
- b\*. at any time, client loses connectivity:
  1. Store all data introduced by the client and inform the client about the respective issue;
- 3a. The platform detects that the data (or any subset of data) entered must be unique and already exists in the system:
  1. The platform warns the client;
  2. The platform allows client changes and re-submission can be done;
- 3b. The platform detects invalid or missing fields:
  1. The platform indicates missing or wrong fields;
  2. The client changes and re-submits;

Special Requirements:

- Client must able to create lambdas on a wide range of screen sizes;



- Textual fields that indicate other system entities like lambdas, dependencies must have auto-complete functionality;
- Node(s) form must indicate information beyond the node name, like where is the node geographically placed;
- Different trigger event fields must appear to the client depending on the selected trigger;
- Lambda code form field must have syntax highlight, syntax checker and auto-indentation;

Frequency of Occurrence:

- Regular

### **3.2.4 Scheduler Use Cases**

The most relevant part, yet invisible, is the system Scheduler, Fig. 24. When a client wants to deploy their project, several use cases must be triggered by the deployer actor. One is the construction of the deployment graph. It gathers all the necessary information of each lambda that one project may have, builds the deployment graph and dispatch tasks across the provider's Runtime nodes. The other use that is triggered at the deployment stage is the construction of the execution graph.

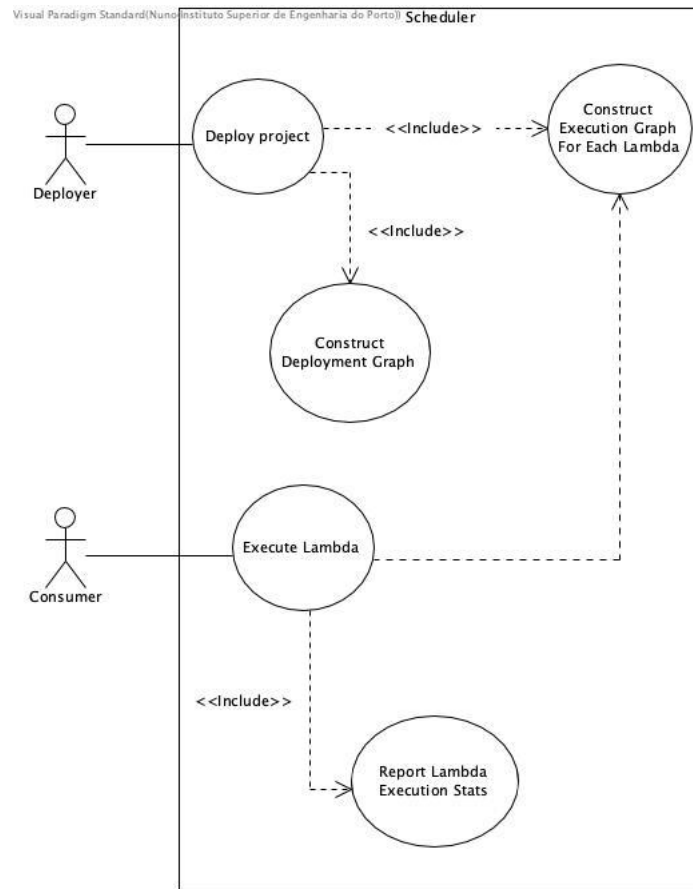


Figure 24 Scheduler use cases

Besides building the two main graphs, it also exports one interface to call one given lambda. It is one of the consumer interfaces available to trigger one graph execution for a given matched lambda. When the consumer triggers one graph, a second use case is included, execution status. Information about the execution is recorded for then be displayed in the Web-Platform.

#### 3.2.4.1 Deploy Project Use Case:

Principal Actor:

- Deployer;

Stakeholders and Interests:

- Client: Deploy lambdas into production; Follow deploy events; Check state of the deployment process;
- Provider: Maximize computational resource usage;

- Deployer: Fault-tolerant, idempotent process to deploy client's lambdas;
- Consumer: Enable consumer with new service(s);

Preconditions:

- Project with one or more lambdas;

Success Guaranteed:

- Project deployment state is updated;
- Project lambdas are ready for execution;

Main success scenario:

1. Deployer actor starts the deployment process;
2. The system request which project is to deploy;
3. The deployer indicate the project;
4. The system computes and stores the deployment graph and execution graph;
5. The system sends the lambda to target node;
6. The system waits for the confirmation;
7. Steps 5 and 6 are repeated until all lambdas are confirmed;
8. The system marks stores the process and informs the actor the result of the operation;

Extensions:

- a\*. At any time, system fails:
  1. The client is informed;
  2. The client restarts the process;
- 4a. The system is not able to communicate with target node:
  1. The system registers this fact;
  2. The system resumes the deployment process, until main flow step 7 is finished;
  3. Categorize affected lambdas;
  4. Stores the process result and informs the user the result of the operation;
- 6a. The system not receive confirmation from the target node:
  1. The system applies a retry policy;
    - 1a. Retry policy succeeds and confirmation is received;
      1. The deployment process is resumed;

- 1b. Retry policy fails;
  1. The system registers this fact;
  2. Continues with the remaining lambdas (until step 7, main flow);
  3. System verify lambda liveness with indirect observations;
    - 3a. Indirect observations indicates that lambda is deployed:
      1. The system saves the process and inform the result back to the actor;
    - 3b. Indirect observations reveal no sign that lambda was deployed.
      1. Affected lambdas are categorized;
      2. Stores the process and responds back to the actor;

Special Requirements:

- Have one or more deployment requests running in multiple scheduler systems do not have affect in the intended result. In other words, the lambdas are deployed with exactly number of instances defined by the client.
- Pluggable retries policies can be inserted at steps 6a1 and 6a1b3 in the extensions flow.

Frequency of Occurrence:

- Regular

3.2.4.2 Execute Lambda Use Case:

Principal Actor:

- Consumer;

Stakeholder and Interests:

- Consumer: Call services(s);
- Client: Offer service(s);

Preconditions:

- Project in deployment state;

Main Flow:

1. The consumer invoke lambda;
2. The system identifies the lambda and the project belonging to;
3. The system verifies project deployment state;
4. The system runs the execution graph;
5. The system collects the results of graph execution and forwards it to the consumer.

Extension Flow:

- a\*. At any time, system fails:
  1. The consumer is informed with the possibility of trying again;
- 2a. The system does not recognized lambda:
  1. The system registers the fact and responds back informing the consumer;
- 3a. Lambda exists but project is not in the correct state:
  1. The systems notify the owner;
  2. The system informs the result of the operation back to the consumer;
- 4b. During graph execution is detected that some lambda host cannot be contacted:
  1. The system executes partial deployment policy:
    - 1a: Partial deployment policy succeeds:
      1. Graph execution is resume to normal execution;
    - 1b: Partial deployment policy fails:
      1. The system halts the graph execution and informs the consumer;

Special Requirements:

- Building execution graph does not have time impact in the overall execution;

Frequency of Occurrence:

- Often

### 3.2.5 Runtime Use Cases

The Runtime, Fig. 25, allows the deployer to fulfill his purpose of deploying a given lambda belong to a specific project. The use case of launch the Sandbox also is used by the actor

deployer and it allocates the resources and launches the sandbox contained the client's lambda. Another aspect is the actor provider has the use case of launch the runtime. The Runtime automatically tries to register into Web-Platform and obtain dynamic configuration such as bus connection parameters.

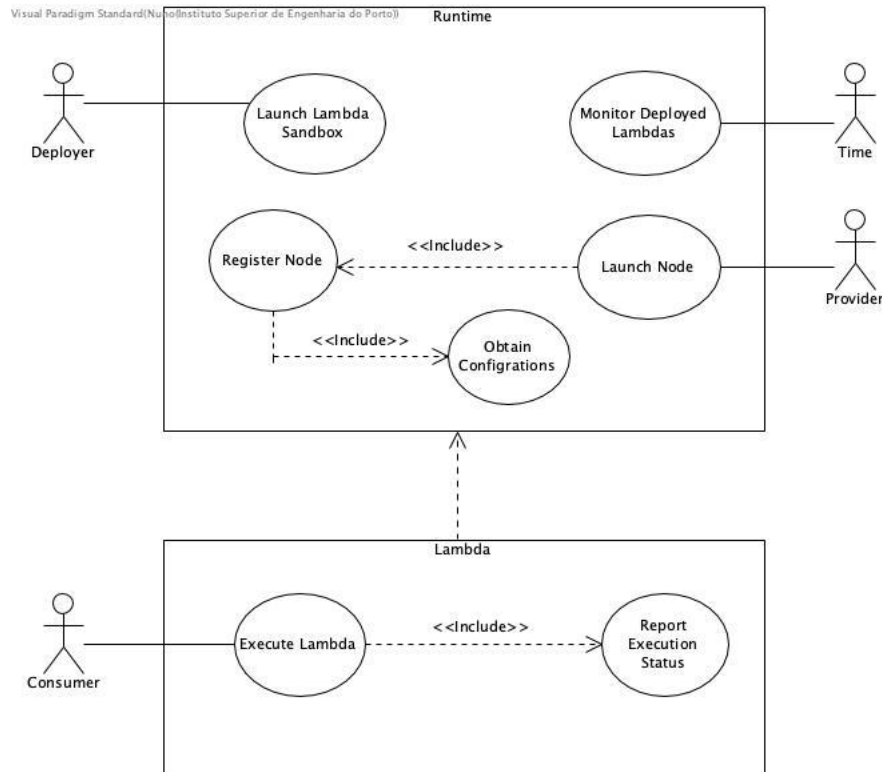


Figure 25 Runtime use cases

The time actor as the important use case, it queries and records resource consumption of the Runtime itself, as well as the deployed lambdas. The records are published to later be displayed into the Web-Platform.

### 3.2.5.1 Launch Lambda Sandbox Use Case:

Principal Actor:

- Deployer;

Stakeholders and Interests:

- Deployer:

Main Flow:

1. The deployer starts the process;

2. The system applies sandbox requirements;
3. The system installs sandbox dependencies;
4. The system starts the sandbox and replies the result of the operation;

Extension Flow:

- a\*. At any time, system fails:
  1. Is up to deployer to try again;
- 1a. The system detects that particular lambda sandbox is already running:
  1. The system response is as if the sandbox was started as fresh;
- 3a. The system could not install dependencies:
  1. The system halts the execution and informs the actor;

Special Requirements:

- Node failures does not have to involve a new deployment;

Frequency of Occurrence:

- Regular

The Lambda system has two use cases, bottom box in Fig. 25. The execute lambda use case, runs a concrete lambda for the consumer actor. The associated report execution status use case is meant to record statistics of the execution associated and as previous reports use cases, it publishes the data to be viewed by the client.

### 3.2.6 Supplementary Specification

The supplementary specification was defined following the FRUPS+ model. Below are the principal points that were identified according to the model and each has a description of how they are related to the proposed solution.

#### 3.2.6.1 Design Constrains

The runtime will run a wide-range number of devices. To have minimal runtimes sized binaries, the logic implemented must be easy, compact and not complex to be easily replicated across multiple programming languages, different software stacks, and hardware architectures.

#### 3.2.6.2 Supportability

The lambda runs on a concept of a sandbox, according to the underline operative system (if any), different types of isolation (Linux control groups, Docker, virtual machine, LXC, etc.) can

or not be used. It's reasonable to offer the possibility of the client download the runtime that reflects the system that the client is targeting to.

### 3.2.6.3 Functionality

The security is also a major point to address, providers with bad intentions could compromise a significant part of the system, especially those who can leak sensible information by performing Man-in-the-Middle attack. The solution must include end-to-end solutions to have some mitigation to the attacks mentioned.

## 3.3 Technologies

The technologies that will be discussed, in great detail, makes part of building blocks of the implemented solution. We also use other tools that will not be discussed because they don't add anything in terms of the overall understanding of the platform, for instance, for the web platform the front-end was implemented in AngularJs, the back end was written in Rust programming language with Actix, a message-based framework model. The Scheduler, due to the single-thread-event-based programming model, Node.js was the appropriate solution. The runtime was written in Node.js for the reason mentioned before but also for having a wide range of pre-compiled binary for other architectures like ARM (arm64, arm6, arm7, etc.), PowerPC, etc, thus, is an easy language to build prototypes and experiment new ideas.

### 3.3.1 Apache Cassandra

Apache Cassandra (Apache Cassandra, 2019) is an open-source database that provides distributed persistent storage, designed to run on commodity servers for managing structured data with no point of failure. It is a NoSQL database, that under this type of databases, it belongs to a column-oriented family. In terms of the CAP theorem (Gilbert & Lynch, 2012), this database guarantees availability and partition tolerance. High availability by replicating data across nodes. Partition Tolerance since every node can act as a node coordinator to perform a read/save operation. Those are the main features to achieve availability and partition-tolerance properties.



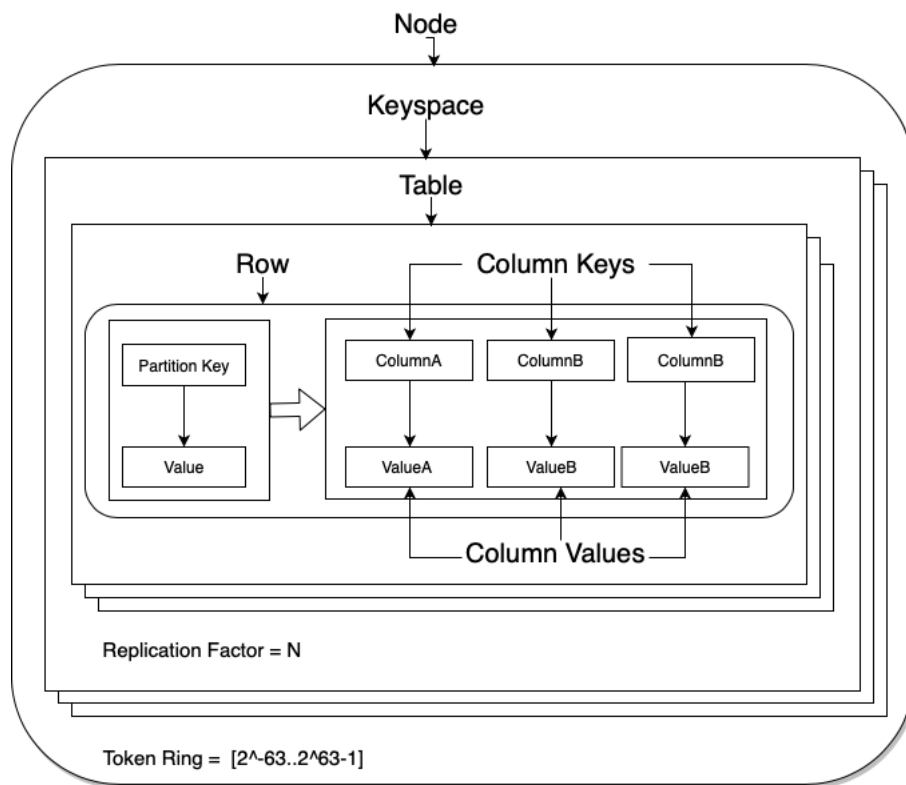


Figure 26 Cassandra entities

The data model has constituted by three entities, Fig. 26, and they are the following: i) keyspace; ii) table; and iii) row. The keyspace resembles on what is the database concept on SQL databases, a structure identified by a given non pre-existence name, that may have zero or more tables, however, it is a namespace and it defines on how the data is replicated. Table entity represents a set of rows, identified by name within the keyspace. Lastly, the row, which is constituted by the row key and their set of strong typed, fixed-size columns.

It is at the keyspace creation that we achieve fault tolerance. It is defined as the replication factor in the replication strategy option. The replication factor of 1, means that the datacenter only has one copy of each row. A replication factor of 1, follows the logic, of no row, can be retrieved if the node where the data is located is down. A replication factor of 3, means that the datacenter has 3 copies on 3 different nodes.

It is available two different replication strategies: i) simple strategy; and ii) network strategy. Simple strategy only allows us to define the replicate factor for a given cluster. The replication is done clockwise (token ring, concept explained later) without taking into account the typology of the cluster. This strategy is recommended to one datacenter, otherwise, the other strategy must be used.

Network strategy defines the number of replicas per datacenters, adequate for business expansion. The keyspace strategy can be altered after its creation, giving some freedom to the developer.

Cassandra uses Gossip, a peer to peer protocol, in which the nodes share information about their state and their neighborhoods. The nodes communicate every second and quickly gain knowledge about their peers and which state there are. Thus, with a helper of a failure detector, Cassandra can avoid routing client requests to unreachable nodes. Snitches are another concept that helps nodes to determine the network topology and it is useful to routing nodes requests, replicas placement (grouped by machines) and determine which datacenters and racks that nodes belong to.

So far, we have addressed the generic overview of this distributed database, now we can explain how the data is partitioned. Cassandra has the concept of a cluster: a set of nodes which map to a single token ring. This token ring has a range of a signed long in any JVM (no exceptions). When two nodes are booted together, let's assume that one node is assigned the range of  $[2^{-63}.0[$  and the other have the subsequent range.

Now that the nodes have a range, it is used the consistence hashing technique to determine where data can be written or read. Every table requires one partition key, is that key that Cassandra uses to apply the one hash function of one of those three implementations: i) "Murmur3Partitioner"; ii) "RandomPartitioner" and iii) "ByteOrderPartitioner". The key hash function produces one finite integer, token value, between the token range. The node responsible for the range that includes the hash value will handle the data request. With those two concepts, token value and token ring, we can describe how this database partition store/read the data. Let's assume the case of Cassandra executing one write with a replicate factor of two. When the request arrives at one node (node coordinator), it applies the hash function, obtains the token value, at this point, it is known which node will host the data and the node stores it. Now enters the replication factor. In this case, is an RF of 2, so, the next node (clockwise) responsible for the next range of the respective token ring, will also store the data.

Partitioners have the responsibility of distributing data across nodes, each node must be set with the same partitioner because they do not produce the same hash values for the same primary key. "Murmur3Partitioner" is the default and not uses a cryptographic function to generate one hash, resulting in 3-5x times improvement when compared to the

“RadomPartitioner” which uses the MD5 hash function to obtain the token value. The “ByteOrderedPartitioner” keeps an ordered distribution of data lexically by key bytes.

The actual data that is persisted in the disk, Cassandra takes a novel approach to increase its read/write latency performance and avoid collisions in middle air. For instances, when two clients override the same row and both send this update to the node, which update is right? Cassandra to avoid this problem, it uses “Write-Before-Read” instead of “Read-Before-Write”. The storage engine groups inserts and updates together, and at intervals, it sequentially write the data in the disk in append-only mode, becoming part of an immutable piece of data, where resides all data. Returning the row, the sequential seek, will find the last row that was inserted.

This technique is leverage by the usage of one special data structure namely Log-structured merge-tree (LSM tree). This data structure like other trees it maintains a key-value pair and allows multiple levels of the same data-structure. Typically, the level  $l : l \in \mathbb{N}$  is merged into the level  $l - 1 : l \in \mathbb{N}$  after a given threshold. The process of merging levels it is given the name of compaction. In Cassandra, each key is the token value and the value represent one row. There is multiple SSTables (each LSM tree level) per table and is the place when compaction occurs.

The SSTables are immutable, insertions and updates are processed as there was upserts and also deletions, marking deletions as thumbstones. During the node lifetime, one partition key can have multiple version of the row with different timestamped versions. When compaction is triggered the node creates one new SSTables with the merged data. The node performance is guaranteed even if it is in the compaction process. Cassandra offers several strategies to merge SSTables, table 7.

Table 7 Cassandra compaction strategies

Name	Strengths	Weakness
SizeTieredCompactionStrategy	Write-Intensive Workloads	Low compaction frequency
LeveledCompactionStrategy	Read-Intensive Workloads	Higher disk I/O usage
TimeWindowCompactionStrategy	Time-Series Workloads	Requires row with TTL.

The SSTables are immutable structures, so, when a read request arrives at the node, it is not created a new structure containing a single row. Another structure residing in-memory is used,

which is denominated by the name of Memtable. Exists one per table and the structure holds the writes up to a certain configured level. Surpassed that level, it is created a partitioned index, mapping tokens to locations on disk and the data is flushed.

From the mention entities involved on Cassandra node write, it is missing the commit log. The commit log is shared across all tables, register all write requests and flushes the correspondent data from a given table when the correspondent SSTable is created.

In counterpart, when a read arrives, Cassandra have to search, at least, in two-node structures, Memtable and SSTable. The search results are then combining, and the read request is fulfilled. The following steps are performed:

1. Verify MemTable;
2. Verify row cache (LRU cache);
3. Verify bloom filter;
4. Verify partition key cache;
5. Locate the data on the disk using compression offset map;
  - a. If partition key is found;
  - b. Or consults partition summary:
    - i. Access to the partition index;
  - c. Obtain compress offset map;
6. Fetches the data from the SSTable on the disk.

Three important aspects were explained, how the database distributes data and what are the steps involved in read and writes. From the CAP, Cassandra cannot be a consistency database, it was not designed and implemented for that either. In contrast of ACID transactions, Cassandra follows Available Soft-state Eventually-consistence paradigm, BASE for short. However, it offers strategies to tune consistency.

When a system sacrifice consistency for availability, a great part of the design is to mitigate the part that was taken away, in terms of CAP. Best effort consistency can be decomposed into two segments, weakly and stronger, table 8.

Table 8 Weak vs strong consistency systems

Consistence	Pros	Cons
Weakly	Lower latency;	Propense to byzantine failures;

	Higher throughput; Minimal resources usage; Faster client response times;	
Stronger	Truthful client responses; Data correctness;	Higher latency; Bandwidth usage; Higher pressure on coordinator node;

Cassandra offers several of tunable consistency options either for read and write operations. The Quorum is calculated and rounded down to an integer number. The mathematical expression:  $Q = \left(\frac{\sum_1^d rf}{2}\right) + 1$ . Where  $d$  is the number of datacenters, and  $rf$  is the replicator factor. The resulting number represents the number of nodes that will be contacted. In table 9, shows the example of 3 datacenters with 3 replication factors in each. Applying the formula  $Q = \frac{3+3+3}{2} + 1 = 5.5$ , rounding down giving the total of 5.

Table 9 Cassandra consistency levels

Level	Read	Write
ALL	9	
EACH_QUORUM	6 (2 nodes per DC)	
QUORUM	5 (Any DC)	
LOCAL_QUORUM	2 (Local DC)	
ONE	1 (Any DC)	
TWO	2 (Any DC)	
THREE	3 (Any DC)	
LOCAL_ONE	1 (Local DC)	
ANY	N/A	1 (Any DC)
SERIAL	5 (Any DC)	N/A
LOCAL_SERIAL	2 (Local DC)	N/A

Consider the example of table 9, in a write request, all the available nodes that owns the row will receive the write request, regardless that consistency level specified. In a write context, the

consistency levels indicates the number of acknowledges that coordinator has to receive from the replicas nodes in order to mark the request as successfully.

If the write request was set with a consistency level of LOCAL\_ONE and 2 of 3 nodes (local DC) where down, the write still succeed, however, the other 2 nodes not received the write request. Cassandra has three built-in mechanisms to deal with this issue:

- Hinted Handoff;
- Read Repair;
- Anti-Entropy Node Repair;

Hinted handoff is an automatic process that is used when the coordinator node could not receive the acknowledge for a write request, even the replica node is down or unresponsive. The coordinator stores the read in one special system table. This table holds the data, during a finite configurable time window on the behalf of the replica. When the node comes online and the coordinator detects it, the data is replayed.

The read repair is the process of comparing the data that comes from the replicas. When a read request is being served. If the data is not the same, Cassandra acts as a consistency level of ALL, requests the row to the reaming replica nodes, compares the rows and the most recent version is sent to the replicas that differ from the more up to date version.

Anti-entropy is a manual process, it makes part of a maintenance plan that the developer must have contemplate when this database is used. Even with the 2 mechanisms mentioned above, data inconsistency can still exist. Cassandra in this mechanism compares all data between all replica nodes. Each replica builds the Merkle Tree per table and then compares the trees between replicas (bubble sorting logic). Cassandra anti-entropy method also can be executed using different strategies: i) full repair; ii) incremental repair. Different behaviors: i) sequential repair; ii) parallel repair; and on specific sites: i) local; ii) datacenter; and iii) cluster.

In the read request flow, Cassandra coordinator have three possible request that can make to one replica node:

- Direct Read Request;
- Digest Request;
- Read Repair request;

The direct read is the act of asking to one replica node for the respective row. The digest request is the subsequent requests that are made by the coordinator to remaining replica nodes, the total of replicas is set by the discussed consistency level. The replica node confirms that the data from the response of the direct request is up to date, and then all remaining nodes also receive a digest request. If the last step results in non-consistency state, a read repair is started at the background.

Wrapping up consistency topic. Cassandra is able to execute lightweight transactions with linearizable consistence. It is implemented under Paxos protocol (Lamport, 1998) with a quorum-based algorithm. We can think of compare and set behavior without a master database or two-phase commit. When used, it not affects other normal read or writes, but cannot have multiple lightweight transactions at the same time, which must be used where is extremely needed.

To sum up, strong consistency can be achieved if the following condition is met:  $R + W > N$ , and eventual consistency with:  $R + W \leq N$ , where  $R$  is the number of consistency level for read,  $W$  the number of consistency level for writes and  $N$  number of replicas. For example, if we write a row with consistency level on ONE, the read must be THREE if we want strong consistency, this if the replication factor is 3 ( $3 + 1 > 3$ ).

### 3.3.1.1 Creating Tables

Consistence hashing have one great advantage in how data is partitioned, however when modeling the business domain to fit into a NoSQL Cassandra database, the design of the partition key for each table becomes a challenge. First, Cassandra denies executing low performant queries, second, we want the data distributed equally across the nodes and third we want to minimize the number of partition reads.

The partition key is defined when the table is created. The scheme to create a table is very similar to the SQL databases. In Cassandra we can define the partition key in three different ways: i) primary key, ii) compound key, and iii) composite key.

Table 10 Primary keys declarations

create table t(a int, primary key(a));	primary key
create table t(a int, int b, primary key(a,b));	compound key
create table t(a int, int b, int c, primary key((a,b),c));	composite key

To finalize, we will discuss the last row of table 10. The primary key is a composite key, where the partition key is “a” and “b” and the flowing are the cluster columns, in this case, “c”. Consider “a” and “b” and “c” of assuming the bits of the representation of the decimal number of [0..7] in binary. This configuration allows having  $2^3$  different rows in 4 different partitions, table 11.

Table 11 Partition key behavior

A	B	C	Partition
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	2
1	0	1	2
1	1	0	3
1	1	1	3

We quickly can conclude if the “a” and “b” both are “0”, and only change “c” in a range of [0..1000], all the rows created in this scenario will be stored at single place. The read per partition will be extremely fast, however, the data is not distributed evenly across the cluster, even if the cluster has an astronomic number of nodes. The technical term of this is a hotspot when no variation exists within the partition key, the data will be placed at one site. Cassandra offers a persistence with no point of failure, but the developer has to be wise when modeling the data.

### 3.3.2 RabbitMQ

RabbitMQ (RabbitMQ, 2019) is an open-source message broken platform. It implements the advanced message queuing protocol (AMQP). Its plugin extension design allows to use several other protocols:



- Simple Text Oriented Messaging Protocol;
- Message Queuing Telemetry Transport;

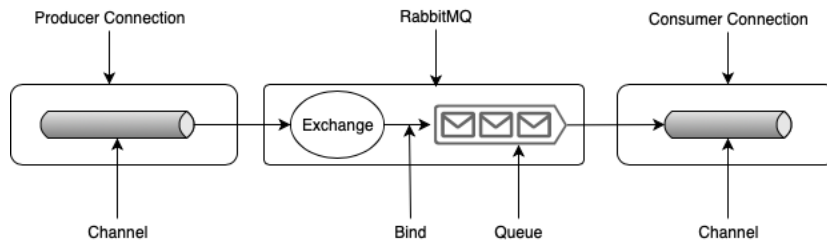


Figure 27 RabbitMQ messaging overview

RabbitMQ is used to deliver messages to consumers. Messages sent by publishers through exchanges, Fig. 27. The consumer is the entity that has one associated queue and the queue has one binding with a particular exchange. The publisher sends messages via exchanges. In other words, when the publisher wants to deliver one message, it delivers it to a middleman who knows how to route that message. When a message arrives, the middleman looks at the message meta-data and checks the address. If the address is a valid routable address, it delivers into the recipient's queue.

Messages are routed and it is done in the exchange. The type of routing dictates how to messages are delivered to the queues. Exists different types of exchanges with different routing properties:

- Direct Exchange;
- Fanout Exchange;
- Topic Exchange;
- Headers Exchange;

A message that arrives in an exchange, declared as direct, goes directly to the queue(s) whose binding key exactly matches the routing key of that message. The fanout exchange copies and route a received message to all queues that are bound to it. Topic exchange, similar to direct exchange, with the difference that the routing key can have wild cards. Messages are routed to one or many queues based on a matching between a message routing and this pattern. Headers

exchange, as a topic exchange, with the difference of routing messages based on the message headers instead of routing key.

Messages have two parts, the message that the producer sent and meta-data. There are no constraints on the actual message, is up to the producer and the consumer to agree in a common format.

In a multi-format ambient, the respective format used can be indicated in the message meta-data, more specifically, in the content-type property. Another related field is the content-encoding, which is identified what compression algorithm was used to compress the message. This is special relevant to reduce messages sizes. Both fields do not have meaning for the RabbitMQ, the actions that those fields can represent is up to consumers implementations to interpret.

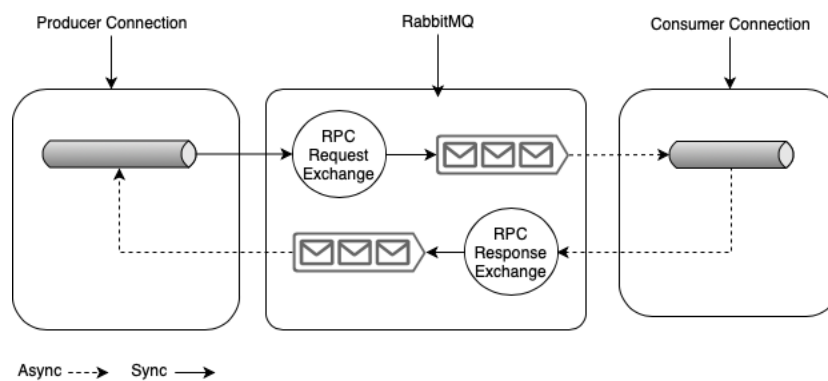


Figure 28 RabbitMQ RCP pattern

RabbitMQ can also be used to implement a remote procedure call system, Fig. 28. The meta-data has two particular fields for this purpose, correlation-id, and reply-to respectively. The design follows the flow: Publisher creates a queue and binds it to a well-known exchange, then, it publishes the message with the field's correlation-id and reply-to (name of the created queue) defined. The consumer processes the message and replies into the queue specified in the reply-to field. The response message arrives at publisher by the consumer publishing the message into the well-known exchange and used the value of the reply-to as routing key. The field correlation-id helps the publisher to correlate requests with responses to further process.

RabbitMQ quickly enables developers to build distributed applications using it as a communication bus. Since the bus can connect multiple applications, geographically distributed

via multiple links as a consequence of network failure, node failures, logic errors may compromise the reliability of the system.

Messages sent via unreliable links is a reason alone to the need of having a mechanism to reactive handling failures. RabbitMQ offers to patterns consumer acknowledgment and publisher patterns.

When the consumer application creates one queue, it can be defined with automatic or manual acknowledgment. As the name implies, automatic acknowledgment, a message is considered successfully delivered immediately after it is sent. If the consumer's TCP connection closes or another consumer-related error occurs, the message will be permanently lost.

Manual acknowledgment, the consumer confirms the delivery tag. A delivery tag is an integer number from a RabbitMQ counter scoped per channel and associated with a message. Three methods are available to acknowledge messages:

- Basic Ack;
- Basic Nack;
- Basic Reject;

The basic ack replies to the RabbitMQ that the message associated with this delivery tag was been delivered and can be discarded. Basic nack and basic reject are used to indicate if the message will be re-queued or discarded. The difference between nack and reject is that nack can reject multiple messages at once. The semantic of basic nack is to re-queue all unacknowledged messages deliveries up to this delivery tag.

The last aspect of this pattern is the number of messages that are pushed to the consumer in an unacknowledged state. This number can be configured by channel or consumer.

Publisher confirms pattern is about the confirmation that the published message was indeed received by the broker. In similarity to the previous pattern discussed, the broker will use the same consumer methods to inform the state of delivery. Remember that a publisher sends messages to exchanges. The exchanges are object managed by the RabbitMQ itself. So, the question that arises is: how can we have security that the message was received by the final

recipients? When we publish a message, a mandatory field can be set. When this field is set, the exchange will return the message to the publisher, if no route exists. Those patterns, when used together can guarantee at-least-once semantic.

This section resumes in how RabbitMQ works. Other features are available such as authentication and authorization, TLS support, cluster and federation support.

### 3.3.3 Protocol Buffers

Protocol buffers (Google, 2019) is an open software tool that developers can use to serialize structured data. The developer defines data structures (messages) in the proto definition file. The protocol compiler named protoc, take those definition files and generate stubs to the desired target language(s). The protocol buffers offer back and forward compatibility and efficient binary format. It was developed by Google to be simpler and faster than XML, currently, it is used for nearly all inter-machine communication at Google.

This technology uses the latter technique to binary encoding. This encoding process uses the top bit in each byte to indicate whether or not there are more bytes coming. The remaining 7 bites of each byte are used to encode the actual number. This technique is denominated by variable-length integer (Varint) encoding and follows under the premise of numbers are not uniformly distributed and the usage of smaller numbers are predominant in computing, so bandwidth can be saved since smaller numbers will need fewer bytes to be encoded.

Table 12 Variable integer encoding

Decimal	Binary	Varint
300	1001 01100	1010 1100 0000 0010
1	0000 0001	0000 0001

Numbers between 0 and 127 only take one byte to be encoded, while between 128 up to 16383 only two bytes are required to encode the numbers in that range. The steps to decode the decimal 300 are the following:

1. Remove the most significant digit for each byte;  
     ➔ 010 1100 000 0010
2. Reserve the groups of 7 bits;

→ 000 0010 010 1100

3. Concatenate the number;

→ 1001 01100

Varint can be concatenated, since it is always clear when a number ends, and another begins. The use of this process allows serialization in any order with, plus back and forward compatibility.

In proto definition file, Fig. 29, the developer can specify the version, currently there are two available syntaxes: i) proto2; and ii) proto3, and both are compatible with each other. Besides the syntax version, the file may have zero or messages, each can have zero or more fields. Fields are key-value pairs and a respective data type.

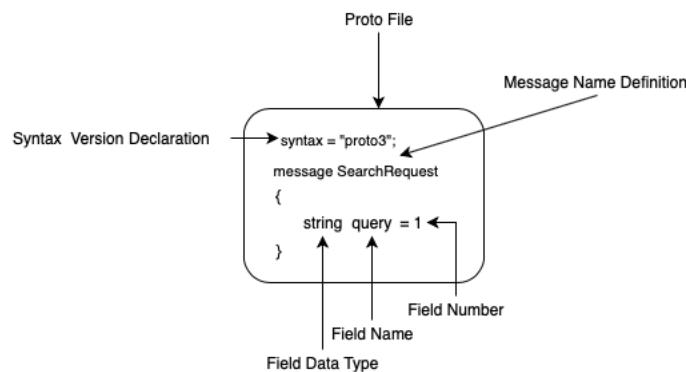


Figure 29 Protocol buffers message definition

One file can have multiple messages and nested messages are also allowed. The field data type can hold all data types that we already are familiar with, such as floats, doubles, strings, byte array, etc. Other messages can be used as data type as well, and more complex constructions like HashMap and enumerations are also available as a valid data type field declaration. If the field is declared as the query field showed in Fig. 29, it is considered as a singular field, default value for syntax proto3, and means that the message can have at the most one field value. In opposition to the singular is the repeated keyword, zero or more with order preserved. Either singular or repeated is specified before of the data type definition.

The field number is required and is used for serialization and deserialization purpose. The field number is a positive integer  $n : \in [1..2^{29}] \setminus [1900..1999]$  and each field has a unique number.

Summing up the theoretical part, this protocol offers interoperability across a wide range of programming languages with strong type accessors. The number of official programming

languages currently that are official supported is 6, however, the great adoption by the community resulted in 39 other programming languages having protocol buffers support.

### 3.4 Design & Implementation

We start to design the web platform. The class diagrams were the first artifact to be produced. Then we have modeled the entity relation model, defined what queries that we want to be able to answers and translate into Cassandra data model. The last step was the REST definition interface.

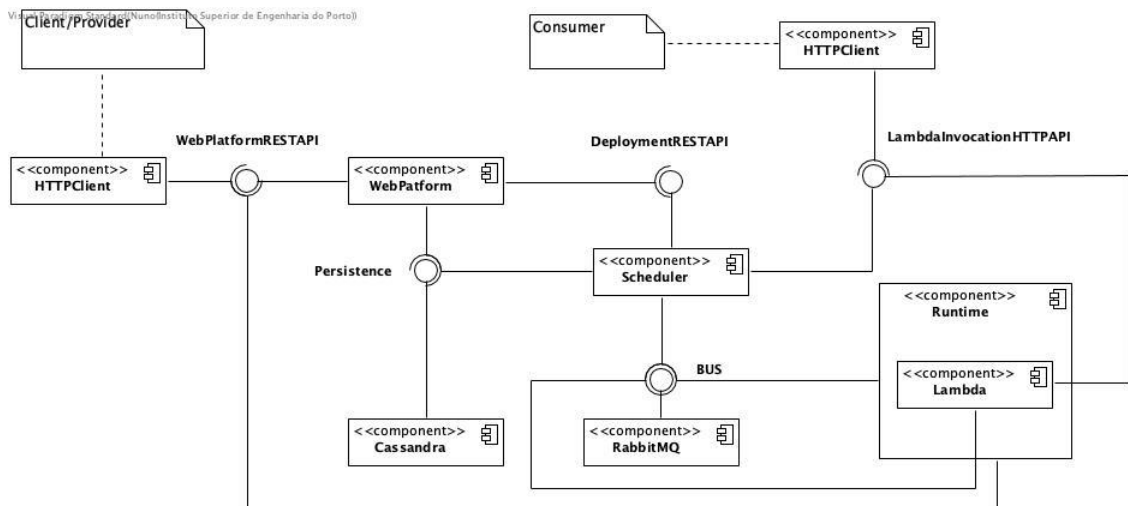


Figure 30 Logic view

The Fig. 30, overviews the architectural view and how the different components interacts between them. The Fig. 30, also has some notes to help the reader to identify the actor that call the which interfaces within each component. We will discuss in depth in the following sections.

#### 3.4.1 Web Platform

In the class diagram presented in Fig. 31, we show all entities and each entity is represented with the respective attributes, and some also have relevant operations, either per instance or class. The data type that we want to point out is the universally unique identifier (UUID) for node designation. The UUID v3 is a namespace name-based, and the specification allows us to use URL's, fully qualified domain names, object identifiers, and X.500 distinguished names. This UUID is used to always generate the same UUID hash for the same node across registrations.

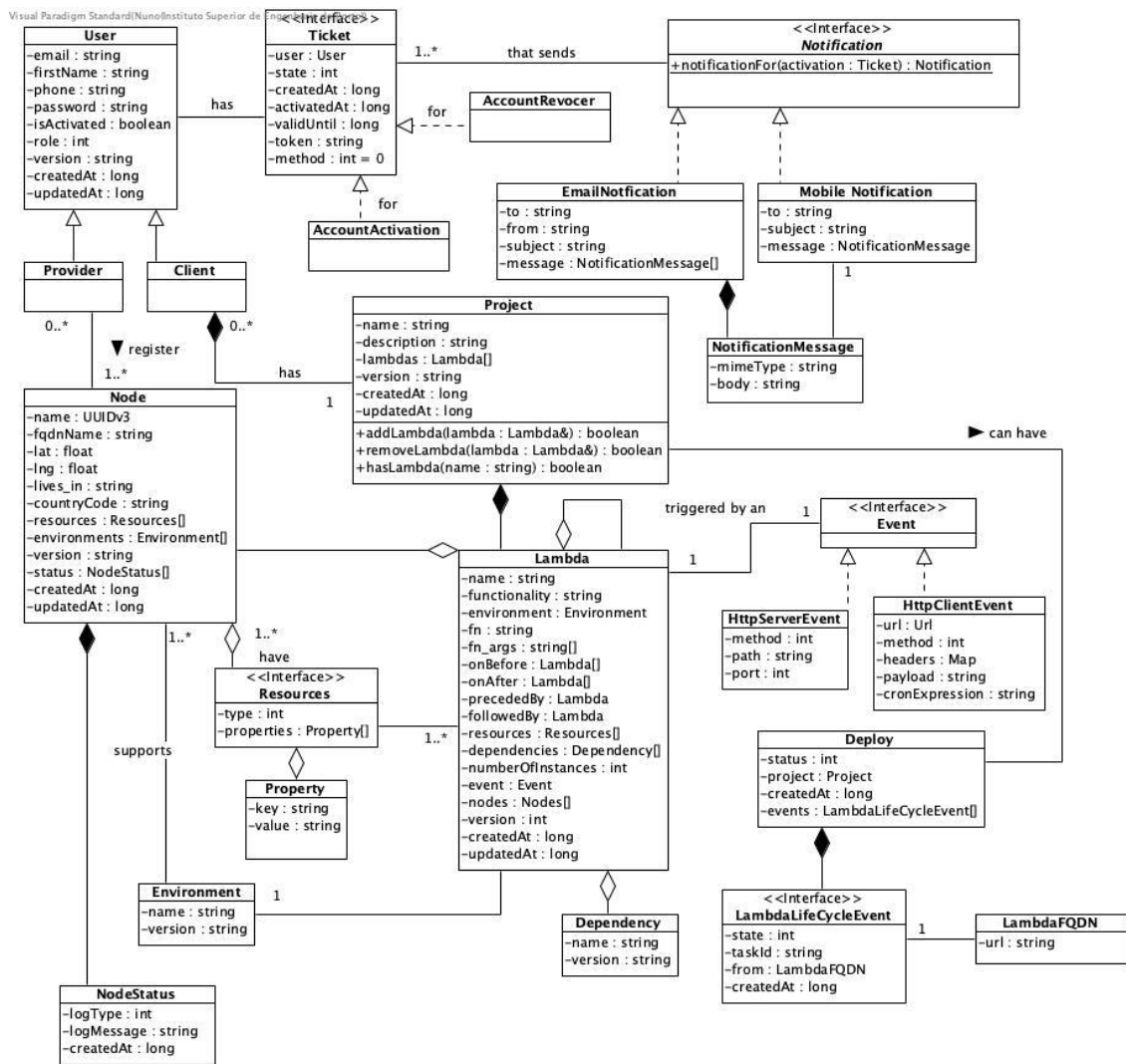


Figure 31 Web platform class diagram

The user has two types of tickets: i) account activation and ii) account recover. The user that does not have a valid account activation ticket does not have access to the deployment process. The users account can be in the possible following states: i) activated, ii) pending activation, or iii) expired. The activation process is a standard flow: the user registers on the platform, the platform sends a notification with an opaque token, and the user send it back. The token can be sent by various communication channels, email and/or phone respectively. For the account recovery, the same flow is used, and the possible states are: i) requested, ii) recovered and iii) expired.

The deploy entity is created by the user for a given project. The deployment process is characterized by the different states that it can be. The states are: i) in deployment, ii) partially deployed, iii) deployed, and iv) failed. In deployment state, is the normal activity of the on-going

lambda remote node setup, installation, and launch. The partially deployed state means that the process was concluded, however not all project lambdas/lambda instances were deployed, nevertheless, the project viewed as a one, it can work normally. The “normal” word refers that the execution graph for all lambdas don’t have any direct dependency we can say that exists at least one lambda instance to work with. The deployed state, as the name indicates, all went right, and the project is running as intended. The failed state indicates that there is not even one lambda deployed in the system because a set of lambdas impacted in the normal execution flow.

#### 3.4.1.1 Data Modeling

We start to model the user entity, the mapping is almost one-to-one in terms of attributes and data types. The only remaining answer is what field to be used as the primary key and the replication factor. We want a unique value that identifies a user in the system and the possible fields were, the email and the phone number field. We took the email since telecom operators recycle phone numbers. Defining the email as primary key, the only query that we want to execute is: give me the user who has this email. The replication factor for this table is 3 per DC, this means that we can have one node down with consistency level of LOCAL\_QUORUM.

The ticket table assembles the two different, account activation and account recovery, entities with the discriminator of “of\_type”. The primary key is the token, and for obtaining the user, it also was added, a column that will contain the primary key of the user table. From this table we want to know if the given token exists, and if exists, who is the user. The replicator factor is 2 per DC.

When the user uses the platform, the top structure that sees is a list of projects and a careful primary key must be designed since its permutation have a direct impact on the query performance. We want to be able to answer: i) give all projects from a given user, ii) give me the project given the user and the project name, iii) how many projects a given user has, and iv) does the project exists from a given user. The order is irrelevant, but we want the answers per Cassandra node. For this, we have used a compound primary key, where the partition key is the primary key of the user table, and the cluster column is the project name, therefore, the user and the user projects will be in the same cluster node. We also want that the system only has one project name across users, the primary key of the project table as we have defined, only guarantees that we do not have the same project name per user. If we change the primary key



to be the project name, the projects will be distributed across the cluster and we not be able to answer: what are the projects by user, even if we brute force by retrieving all project and filter by user ownership, multiple round trips between the cluster nodes and the contact point to only get all the projects, considering a cluster with 100 nodes, the probability of all be contacted is almost 1, consequently the Cassandra cluster could collapse. To address to the unique project name across the system, another table was created to just answer the question: exist any project with this name.

Lambda	
PK	<b>((belongs, project_id), id)</b>
	id : UUID name : Text functionality : Text environment : UUID fn : Text fn_args : Set<Text> on_before : Set<UUID> on_after : Set<UUID> preceded_by : UUID followed_by : UUID resources : List<UUID> dependencies : Set<Text> number_of_instances : Int nodes : Set<UUID> event_type : Int etag : Text project_pk : Text belongs_pk : Text created : Timestamp updated : Timestamp

Figure 32 Lambda Table

The primary key of the lambda table, Fig. 32, have the same performance concerns as the project table primary key. First of all to answer: i) what are the lambda(s) from a given project that belongs to some user, ii) how many lambda(s) does a given project have per user, iii) give me one lambda that have some id and belongs to some user and to the this project name. Now, we have another strategy to form the primary key, Cassandra allows to split the partition key into parts, this is named composite primary key. We want a single round trip per project lambda(s) that belongs to someone. The lambda(s) will not be stored at the same node as the project and user table, but the lambdas will be stored at one single node. The reason for figure 31, besides being the lambda table definition is also to show the different data type that

Cassandra allows. For instances, the lambdas that run in parallel (on\_before) is as defined as a set data type, and from the set theory, a set is a collection of non-repeated elements. More complex data types are also allowed, such as Maps, inet addresses, and user-defined types. The dependencies of the lambdas were integrated into the lambda table, each element in the set is a string where the dependency name is concatenated with the version, for example, amqp@2.0.

The event triggers are a point of variation, although we only have represented two of them, we need a flexible and easy way to integrate new types of event triggers with different properties. Where considered the possible options to tackle this variation: i) create a table with all known columns (aka single table heritage), ii) create a table for each event type and iii) use a single table using entity-attribute-value modeling. The downside of the first approach is sparse columns followed by a possible alter table command to accommodate new attributes. The second option is an easy way at the cost of a table creation per event, with one master table containing pointers for all available event tables. To answer the questions: give me all available events, the number of the database queries that needed to be executed is  $q = 1 + E$ , where  $E$  is the total number of event tables and the 1, is to get all event tables from the master table, plus two round trips between client and the contact point. We used the EAB model, a single table, this allows adding different events with different attributes on a single table discriminated by the entity (event). The primary key is a compound key with the partition key being the event type, a numeric value, and the cluster columns of attribute and value.

The node table also is a mapping one-to-one and the complex types have their own table. The answer that we need from this table is: give me all nodes from a particular country. The compound key is constituted by country code being the partition key and the name is the correspondent cluster columns, with irrelevant retrieval ordering. This theoretical allows equal node names across different country codes.

To reduce the complexity, we have created two resource tables, standard resources table, and resources table respectively. The user can choose multiple nodes to deploy a single lambda, the creation lambda form will display the row of the standard resources table (e.g. 1 CPU, 2 CPU, 120MB RAM, etc.), in background the form queries the platform if the resource(s) that user has chosen are available from the selected nodes, this is done by querying the resource table.

The environments have a similar approach. We have one table, execution environment table, that have the compound primary key: name (e.g. NodeJS, Python, Java, etc) and version (e.g 10.0.2). The execution environment table is from the client selects the desired environment to code the lambda logic and this table is populated from the inverse execution environment table. The inverse execution environment table is populated when the provider registers a node and have the fields: name, version, and nodes (set). This design lets us know what are the nodes that implement one specific environment, this makes form validation easier, the downside is the extra logic in the platform to keep those tables in sync.

The node status table is one of the tables that retrieving rows in temporal order matters. This table has a compound primary key where the node name is the name of the node and the cluster column is the timestamp. The clustering order is descendent, we want the older rows first.

The deploy table is almost a one-to-one mapping, where the compound key is the primary key of the user table and the project name of the project table. The deploy will be in the same node where the user and project is. We can answer give me all deployment(s) that particular user has. For the last, the life cycle table resulted in three equal tables with different primary keys arrangements to answer different questions. A lambda execution can be seen as a Linux process, where every process has a thread group id, every spawned thread belongs to that thread group, in other words, when the lambda is executed, every lambda(s) involved must have the same id, in our case a "taskId". The life cycle table must answer to three questions: give me all executions that particular lambda had in descent chronologic order, ii) give me all processes in descent chronologic order, and iii) give all deployment life cycles per project in descent chronologic order. The last is obvious, we want to show to the client what has occurred in a deployment cycle, for example, what pid was attributed to the lambda, or what was the output of the dependency manager, etc. The other two are complementary yet different, with this we have a drill down and drill up per taskId. The client can see the logs from a specific lambda, focus on a particular execution and drill up to see all process, and another way around is also possible.

Table 13 Database tables

Table	Primary Key	Replication Factor
User	Email	3

Ticket	Token	2
Project	(User_PK,Name)	3
Project_User	(Project_Name)	2
Lambda	((User_PK,Project_Name), ID)	3
Available_Events	(entity, attribute, value)	2
Events	(Lambda_ID, attribute, value, type)	2
Node	(Country_Code, Name)	3
Resources	(Type, ID)	3
Standart_Resources	(Type, ID)	2
Execution_Environments	(Name, Version)	3
Inverse_Execution_Enviroments	(ID)	3
Node_Status	(Name, Created_At)	2
Deploy	(User_PK, Project_Name)	3
Execution_Event	( (Project_Name, Task_ID), Created_At, ID)	2
Execution_Lambda_Event	( (Project_Name, Lambda_ID), Created_At, Task_ID, ID)	2
Deployment_Event	((Project_Name), Created_At, Task_Id, Id)	3

#### 3.4.1.2 Platform REST API

Concluded the class diagram and the respective persistence part was time to design the client's interface. To not turn this part repetitive, we decided to show only three resources and at the end one summary table with all resources available. The illustrated cases cover all implementation aspects that were considered at the designing phase.

The user resource, Fig. 33, have three methods implemented: i) GET, ii) PUT, and iii) POST. Following the REST architecture and the semantic associated with their respective methods, the GET request, in this case, retrieves the user entity. In the GET request, for all resources, is obtained its entity-tag (RFC7232) in the header's HTTP response section. The entity-header field can be used for caching purposes or for optimistic concurrency control (our case).

In our interface, all PUT HTTP commands, require the indication of the entity tag value in the form of conditional requests. This design is an extra step to avoid mid-air collisions (Nielsen & LaLiberte, 1999).

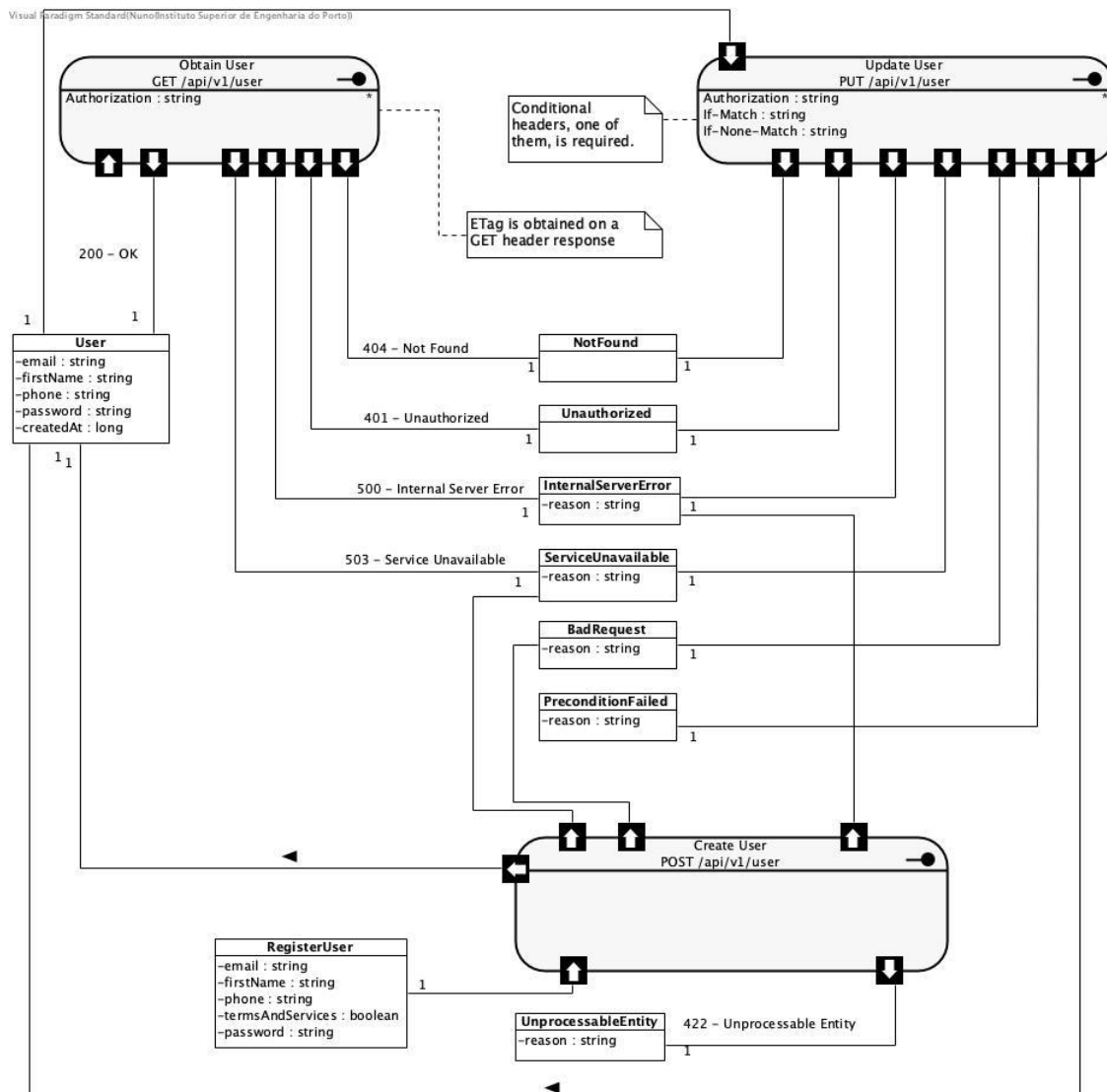


Figure 33 User Resource

We have used the OAuth 2.0 Authorization Framework (RFC6749) to authenticate the user agent, using the JWT Bearer scheme (RFC7523). All clients and providers resources are covered by this authentication, even for HTML resource pages (AngularJS application is partitioned into components that the user-agent requests in the background to the server).

In term of responses, all resources can respond with internal server error or service unavailable. Internal server error is a logic error (uncaught exception, etc.) or a syntax error, this kind of error are not frequent, nevertheless, it can occur. The service unavailable means that the platform is working but some third-party system is unavailable, and the platform cannot complete the request.

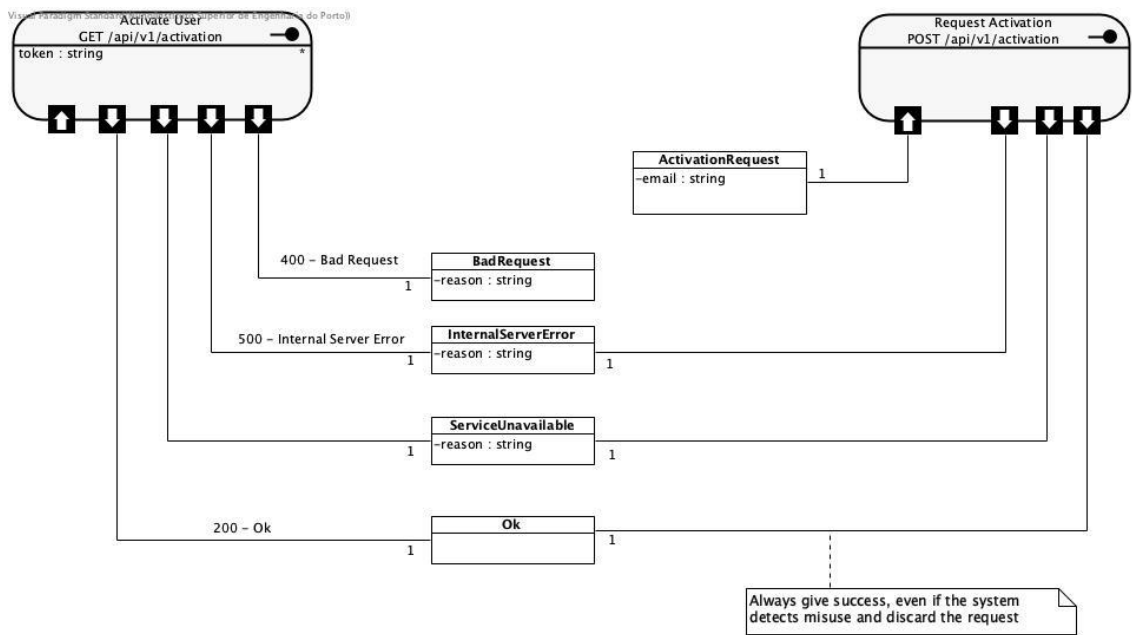


Figure 34 Activation Resource

The activation resources, Fig. 34, have two methods, GET and POST. The first method is to activate one account given one token. The account activation token is received either by email or via a mobile notification. The token can be requested using the POST method to the same resource indicating the respective notification channel.

The last resource, that will be mentioned, Fig. 35, is one resource that requires pagination. Pagination is the act of retrieve incremental parts of a set. The set in our case resides in Cassandra. The driver that is available for Rust does not include pagination, yet the Cassandra binary protocol (Hobbs, 2019) has this feature, is up to the driver implementors to make it available to the users (developers).

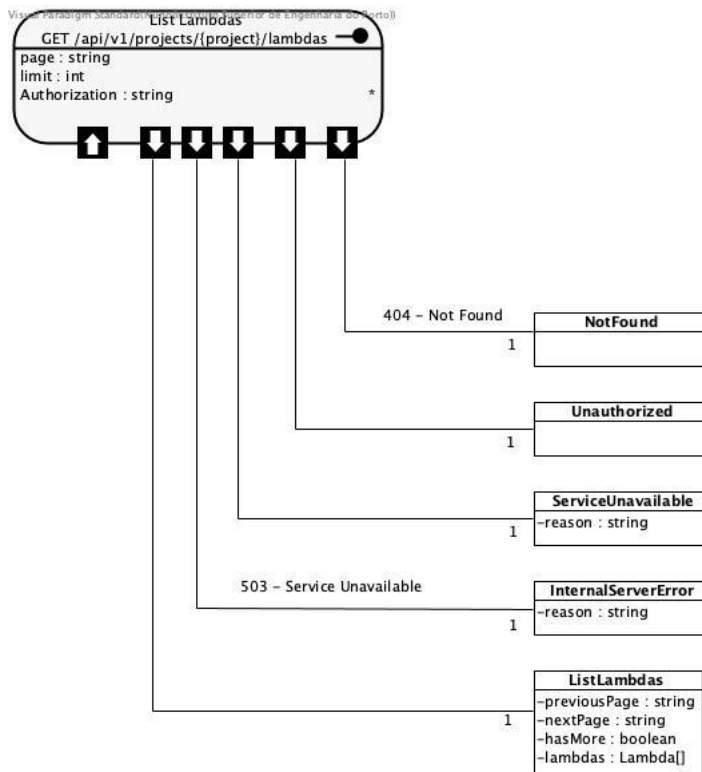


Figure 35 Lambda resource

Not having pagination was not an option, so we have implemented the pagination within the driver (internal state), but also, pagination for stateless connections, using Base64 encoding and decoding technique for safe binary communication (Gonçalves, 2019). We have done the pull request back to the Cassandra Rust repository and the changes were accepted by the repository maintainer. Some important resources needed this feature, like paginate lambda processes, nodes and so on.

Table 14 Platform REST resources summary

Resources ULRs	Methods
/user	GET, POST, PUT
/activation	GET, POST
/auth	POST (login), DELETE (logout)
/forget	POST (password recovery)
/reset	POST (password redefinition)
/projects	GET, POST
/projects/{project}	GET, PUT, DELETE
/project/{project}/executions	GET (logging)

/project/{project}/deployment	GET (logging)
/project/{project}/ executions/{lambda}	GET (logging)
/projects/{project}/lambdas	GET, POST
/projects/{project}/lambdas/{lambda}	GET, POST, DELETE
/nodes	GET
/nodes/{node}	GET, PUT, DELETE
/nodes/{node}/status	GET
/deploys	GET, POST
/deploys/{deploy}	GET, POST, DELETE
/environments	GET
/resources	GET
/dependencies	GET
/probe/projects/{project}	GET
/probe/users	GET
/protocol	GET
/bus	GET

The table 14, shows all available resources that are exported. The Swagger documentation was generated from the Visual Paradigm REST diagrams and was integrated into the web platform. The two last two resources make part of the provider's nodes interaction, and their purpose will be further explained.

#### 3.4.1.3 Platform Graphical Interface

The forms have a set of indicators to make easier its utilization. In Fig. 36 is represented all features that the forms across the platform have. One form normally has one or more fields. The fields can be designed to accept valid datatypes, valid formats and valid lengths, these front-end validations do not discard the correspondent back-end validations.

In the platform, each different validation has their correspondent error message. The user if introduce the name with white spaces, we want to express the error in the more specific way, instead of just saying, invalid name or another generic message.



We want a unique email across the platform, for this, two validations were placed. One is when the client submits the form, another is when the client focusses out the email field. When the email field losses focus, an asynchronous validator will probe if the email already exists, Fig. 35. If the probing process detects a violation, the form will display the correct error response. This technique is also used when the client is creating a new project, more precisely, validating the project name field.

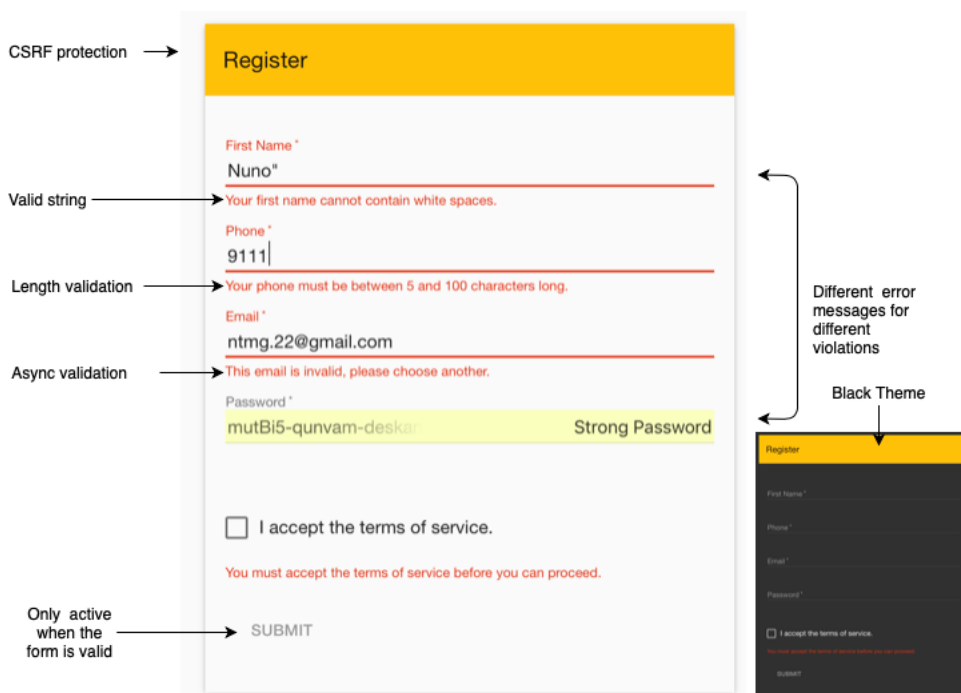


Figure 36 Form features

To reduce back-end work, all form submit buttons are disabled by default. The button is enabled only if the form is valid, this does not mean that is correct but represents a valid back-end request. Another small feature is configurable themes. We let the user choose between the dark and white theme.

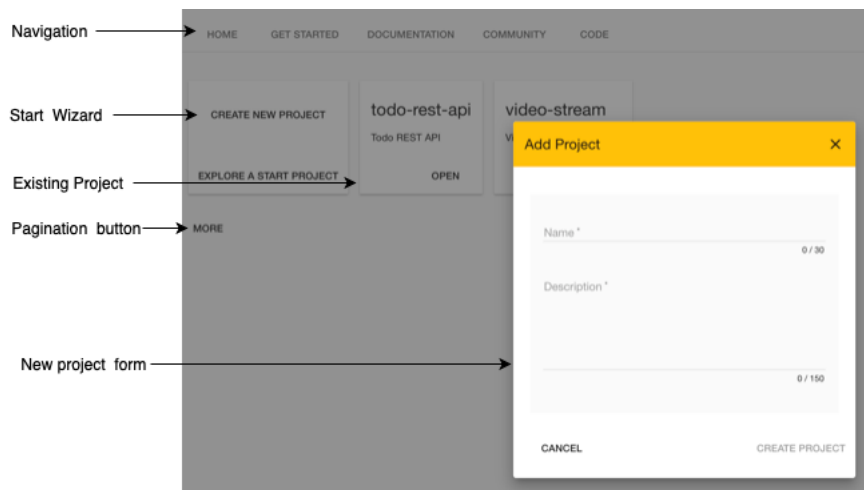


Figure 37 Platform console overview

In Fig. 37, we can see all the projects that one user has. This example is using a paginator limit of two elements and the user can navigate by using the pagination button(s). In start wizard box, the user can create a new project. The dialogue windows contain the form with the respective fields associated with a project. When a user clicks on the project, the view is something like Fig. 38. The account is on the navigation menu and is only shows up if the user clicks on the top right corner.

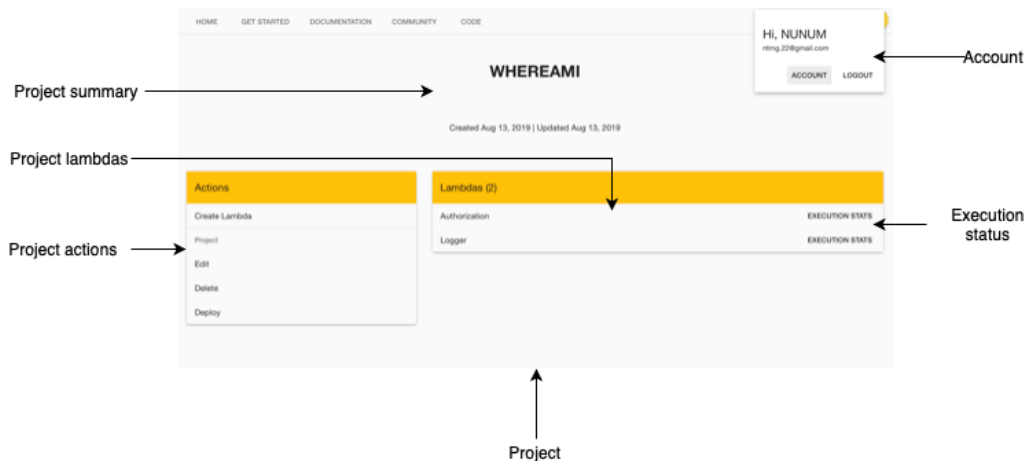


Figure 38 Platform project view

Now we will create a “logger lambda”, and all steps associated, figures 38-42. The page has four forms that work together to send valid requests to the server. The first is the general characteristics form, Fig 39, and as we can see, three important fields are defined at this stage: i) the environment, ii) the node, and iii) event trigger. The environment selection has a side effect on the code editor syntax validation, Fig. 39.

**General Characteristics**

Name \*  
logger

Functionality \*  
Example

Number of Instances \*  
1

Resources  
1, 128MB

Environment \*  
Javascript

Deploy into nodes \*  
zion(fog)

Type of Event \*  
network - HTTP server

Figure 39 Platform lambda general characteristics form

The client also has the possibility of defining the function arguments as a string data type, and the dependencies that this lambda needs. In this case, the “morgan” dependency, which is a logging library, is just to exemplify the process of creating a new lambda. The API that is exported and injected with the name “context” as lambda argument, already have a logger built-in. The API will be discussed at the runtime section.

**Lambda Function**

Function Arguments

Dependencies  
morgan(1.9.1) X

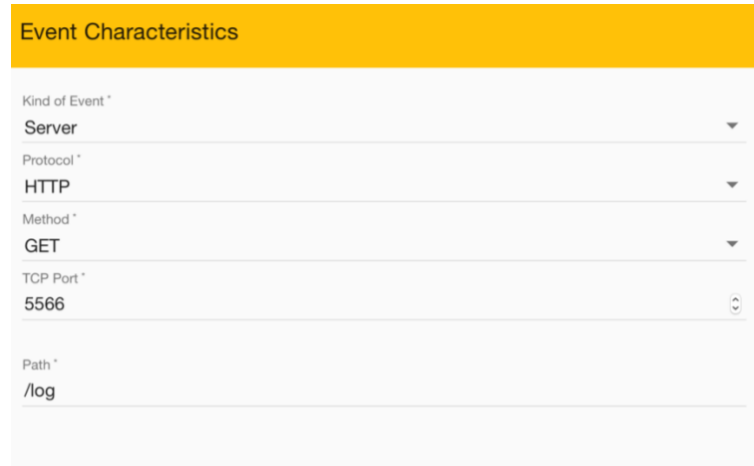
```

1 function main(context)
2 {
3   context.logger().info("execution starts here");
4   return {status:200};
5 }

```

Figure 40 Platform lambda function form

The logger lambda is triggered by a networking event and the correspondent form is in Fig 41. The path can also have parameters to later be extracted, for example `"/log/:name"`, when a client calls the lambda with the path `"/log/warning"`, a map data-structure `"name->warning"` is injected in the lambda arguments.



Event Characteristics	
Kind of Event *	Server
Protocol *	HTTP
Method *	GET
TCP Port *	5566
Path *	/log

Figure 41 Platform lambda network event

One important aspect is to define the execution flow. Exist the form that manipulates the graph visualization, Fig 42. The logger lambda is the target lambda and the authorization lambda that is executed in parallel without interfering in the target execution as previously explained.

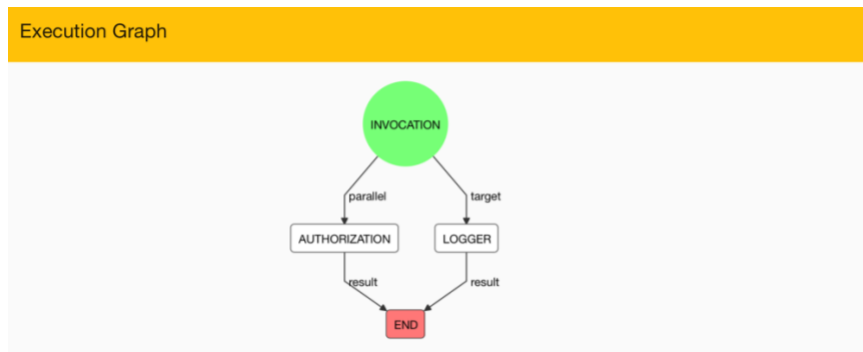


Figure 42 Execution flow

Created the logger lambda, is time to deploy it into the fog node. On the project page, the client can consult the state of the deployment, Fig. 43. In this case, as we can see, it is a re-incident deployment because the logger lambda log states the lambda already have been running.

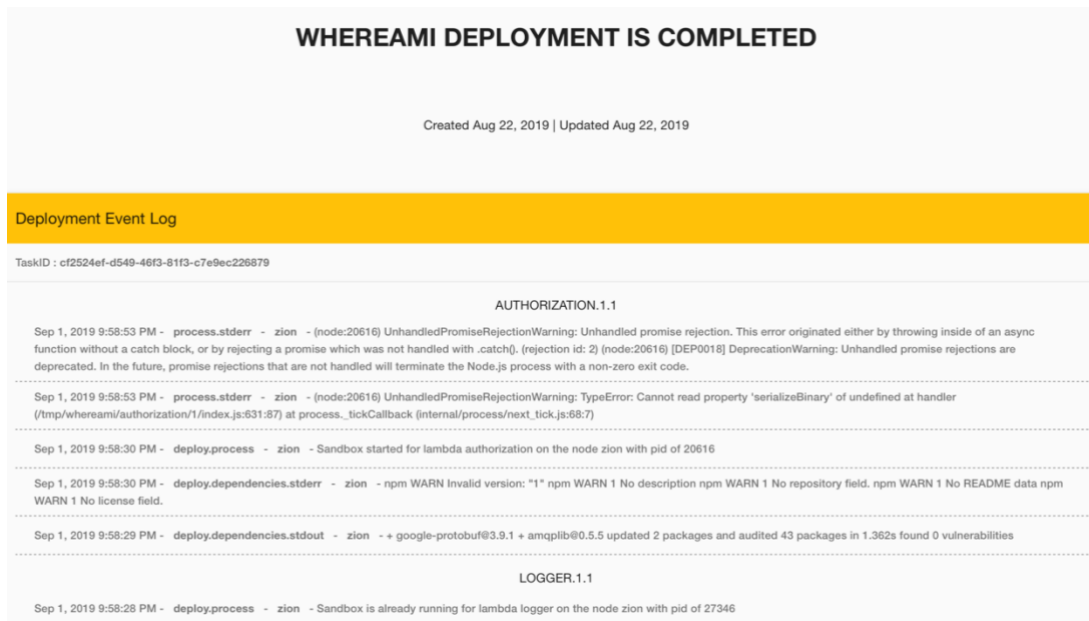


Figure 43 Deployment event log

Deployed the lambda, it is time to see the execution status. In figure 44 is the lambda correspondent executing logging. We can see that it starts with our logging entry defined in the lambda logic. This page also has 3 additional graphs to monitor in real-time the CPU, RAM, and bandwidth usage from each lambda.

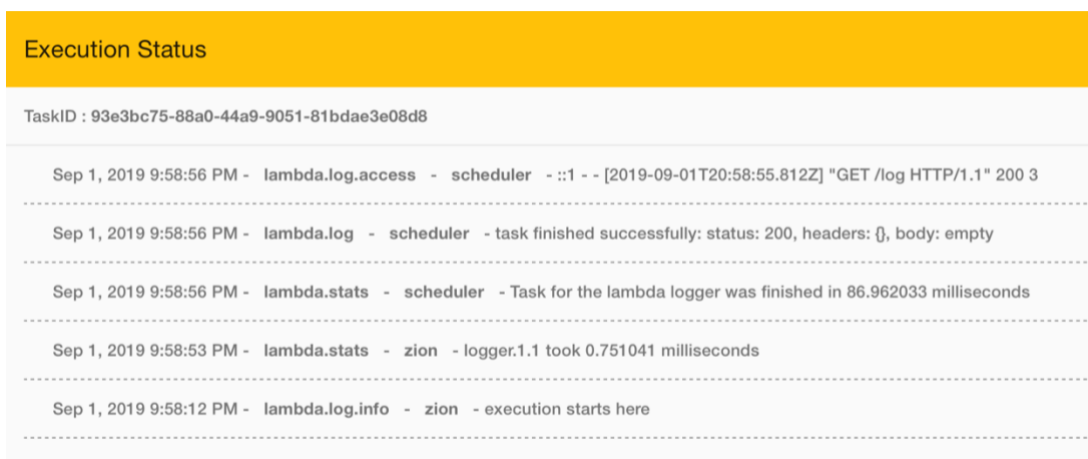


Figure 44 Logger lambda execution log

To conclude, we have shown the principal client uses cases and how the implementation was conducted. The development has the objective of offering a convenient way for the user interacting with the platform.

### 3.4.2 Scheduler

The plugin-based design, Fig. 45, was adopted to handle failures. The failures are associated with unrouted messages. The default implementation follows the extension flow of the execute a lambda use case.

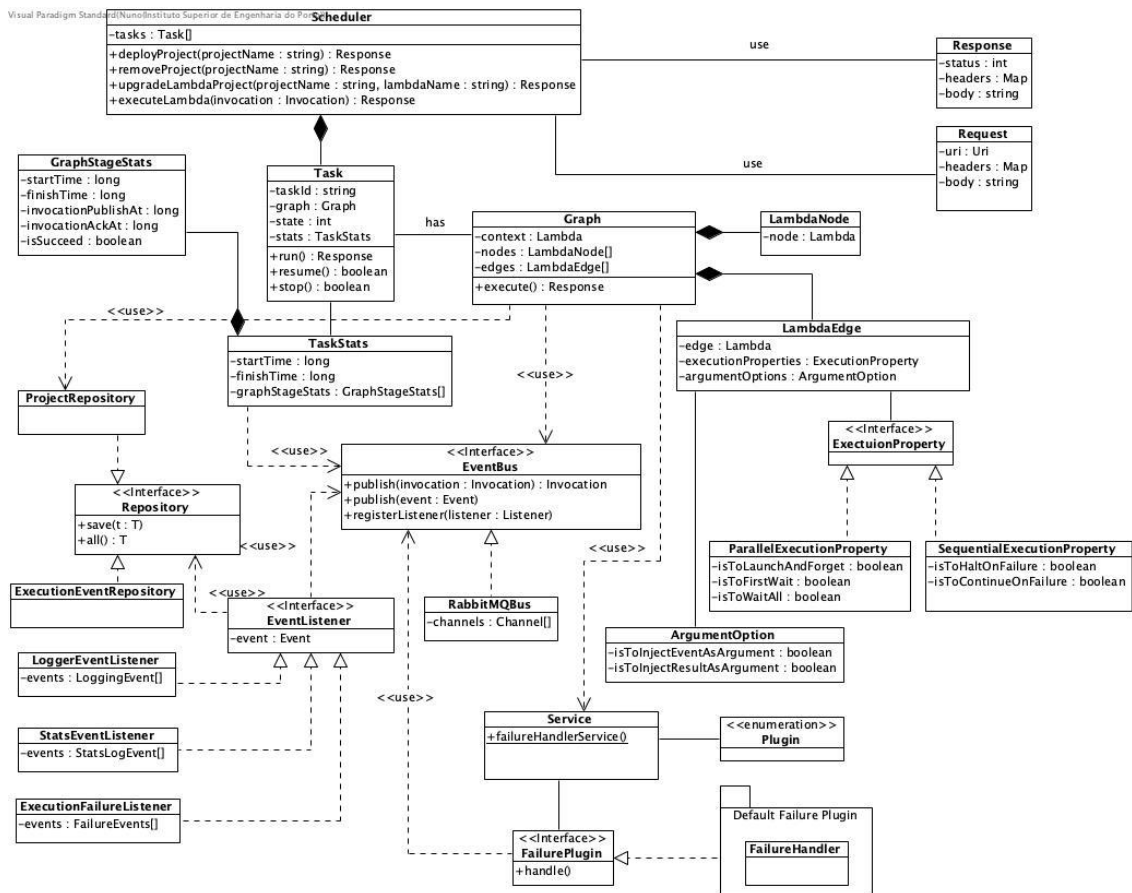


Figure 45 Scheduler class diagram

#### 3.4.2.1 Scheduler HTTP interface

The scheduler exports a REST interface to process project deployment-related requests, Fig. 45. The authorization is taken into account the JWT claims, in other words, no user without the respective permission can trigger a deployment.

The deploy entity handles with 3 different commands, Fig. 46, and they are the following: POST, PUT, and DELETE. When a client requests a new project deployment in the platform, the platform sends a POST command with the respective payload to one of the available schedulers. When a client wants to remove a project, a delete command is executed, at last, to upgrade individual lambda(s), a PUT command with a list of lambdas that the client wants to be upgraded.

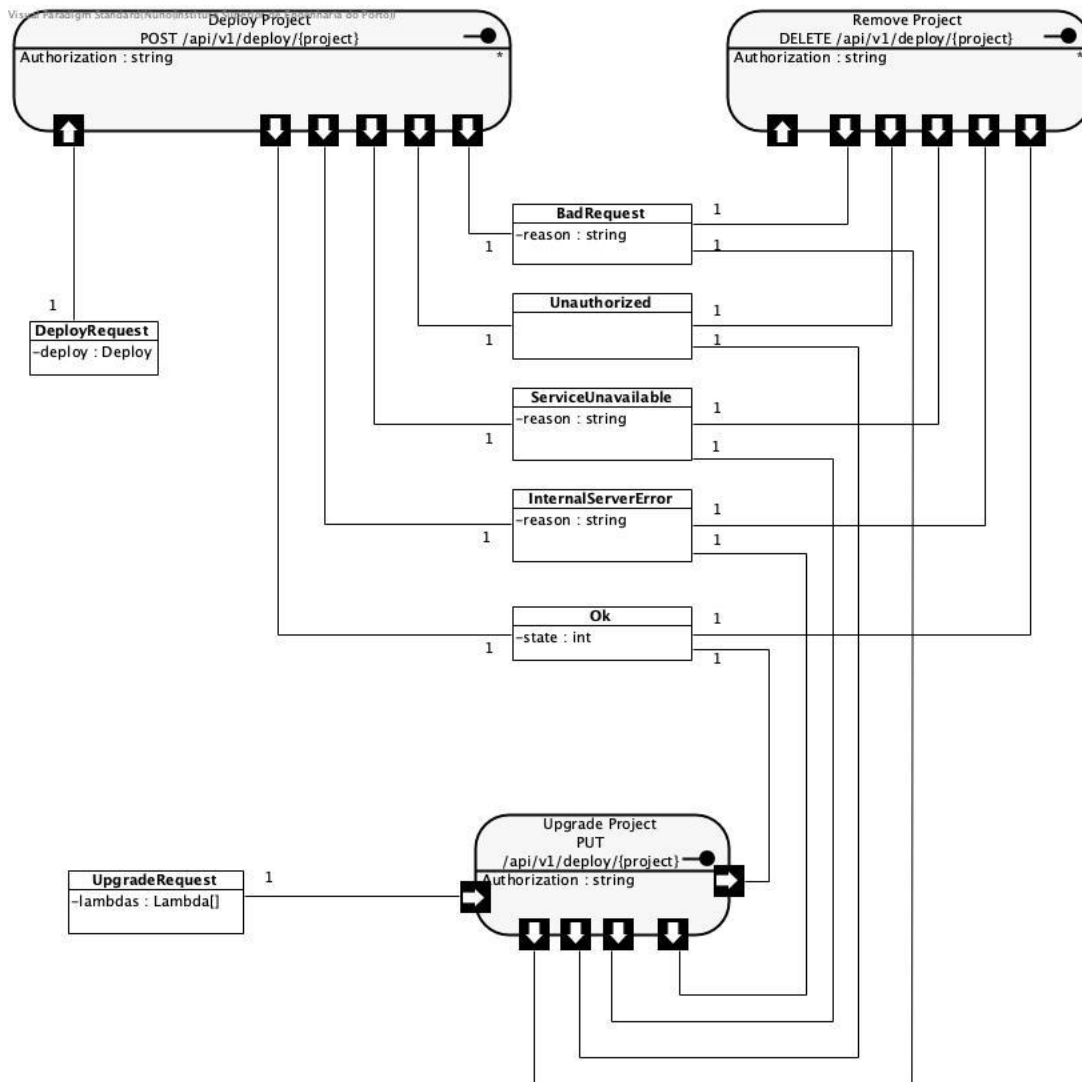


Figure 46 REST Deployment entity

The scheduler has the capacity to coordinate lambda executions, for that, the scheduler has a generic HTTP component that accepts all consumer requests. The component has the capacity to handle four HTTP methods: GET, POST, PUT, DELETE.

When a consumer HTTP requests arrive at the generic HTTP component, Fig. 47, the request is wrapped into a Request instance and is passed into the scheduler component. The scheduler component finds the correspondent project, filters the lambdas that are associated with the consumer HTTP method. The result from the filter operation is all lambdas that were defined as server network event and the HTTP method is equal from the consumer HTTP method. Next, another filter is applied to reduce the set to a single lambda that matches with consumer HTTP request path.

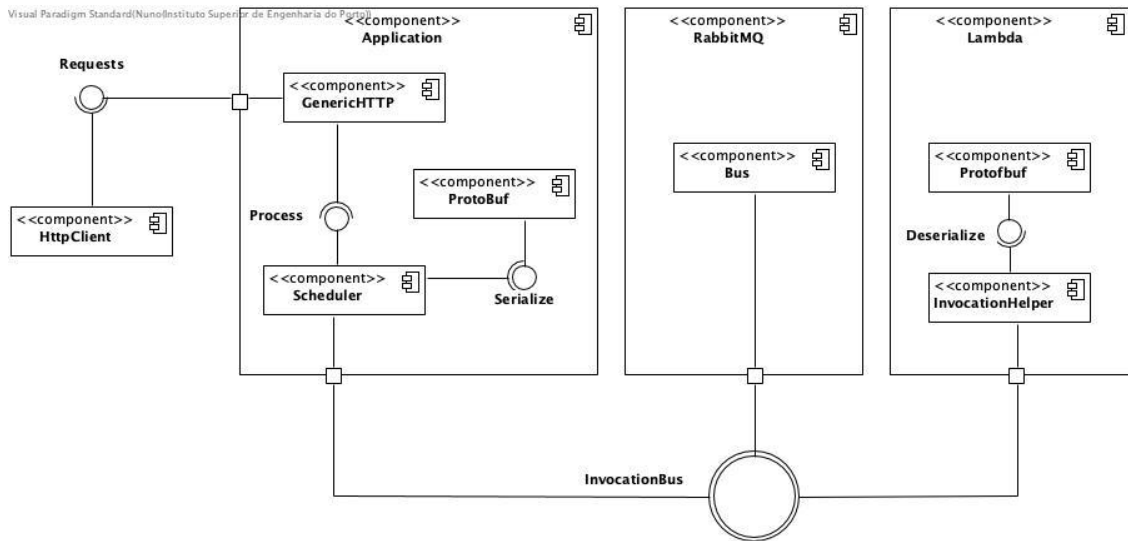


Figure 47 Lambda execution

The scheduler component when finds the target lambda, executes its correspondent graph. For each lambda on the graph is created one Invocation instance that is serialized using the protocol buffers component and is sent to the respective worker node via a bus. The bus integrates all lambdas in the system and each invocation follows an RCP RabbitMQ pattern as discussed earlier.

### 3.4.2.2 Networking

Any scheduler can coordinate any lambda execution from any project. The first answer is how the consumer calls the right lambda and how the scheduler finds the right lambda graph. Note that we do not have any constraint in the path on the lambda server HTTP event, but we do in the project name, the project name must be a valid ASCII string. The project name is used to create a subdomain. The platform is under a domain, and the projects, when a deploy is made, a new subdomain is created in the DNS table. Consider the project “whereami”, when the deployment is requested, the scheduler will create a new CNAME DNS entry, so if the platform is in the domain “density.io” the consumer can call the logger lambda as “whereami.density.io”. This is an easy form of pin down the project, besides each project can host its own SSL certificate, protecting the client and the server communication.

### 3.4.2.3 Bus addresses Setup

The bus integrates all the lambdas, for that, each project has an exchange with the name of the project. All project lambdas are under these exchange with the respective queue-routingKey binding pairs. The scheduler to send a message to a specific lambda, it needs to know the



project name (exchange) and the lambda id and version (queue name). For example, to call the whereami logger lambda, the scheduler makes a call like: “publish(‘wheremai’, ‘id.version’, ...)”. When exists more than one consumer for the same queue is applied round-robin delivery strategy. When a lambda starts running, the runtime passes as process arguments the required information to establish bus communication and to create and consume from a specific queue. The scheduler when publishes the message defines an expiration for each message sent and if any error occurs, the failure plugin will handle the error in the best possible way.

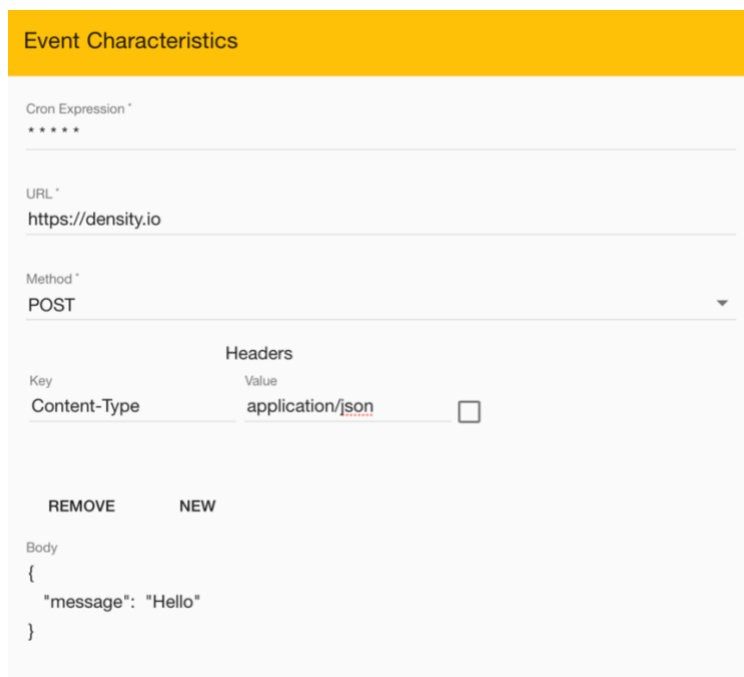


Figure 48 HTTP client event trigger

When a lambda is of type HTTP client, the runtime is responsible to trigger that event respecting the defined CRON expression. In figure 48, the lambda will be triggered every minute with the response of the request to the particular domain. Another well-known queue will receive the response of the lambda invocation response to further execute its execution graph.

The deployment is also done via the bus, in this, the runtimes when starts a UUID v3 MAC address is obtained and the binding is created. The biding (queue->nodeName) is under the deploy exchange. In the whereami example, if the client wanted 4 logger lambda instances in 2 different nodes, the scheduler will sort the nodes by name, apply modulus by 2 (two possible nodes) and call something like this 4 times: “publish(‘deploy’, ‘nodeName’, ...)”.

The logging events have a well-known exchange, all scheduler instances consume from the same logging queue and persist those events into the database. This queue is created as durable, which means, if there is not consumer, the bus will retain the message until one consumer become available to process the message(s).

#### 3.4.2.4 Protocol Buffers messages definition

The binary protocol was defined in 10 different messages using the protocol buffers definition and it is used to publish messages into the bus. The messages can be subdivided into 3 groups: i) deployment, ii) execution and iii) logging.

```
message Command {
  enum Type {
    DEPLOY = 0;
    REMOVE = 1;
    UPGRADE = 2;
  }
  Type type = 1;
  oneof command {
    DeployLambda deploy = 2;
    RemoveLambda remove = 3;
  }
}
```

Figure 49 ProtoBuf Command message definition

The command message, Fig 49, is constituted by two fields and it is used on the deploy exchange. This first field is to define the command type and the command field can be one of deploy or remove lambda messages.

```

message DeployLambda {
  string task_id = 1;
  string id = 2;
  string name = 3;
  string project = 4;
  string owner = 5;
  uint32 number_of_instances = 6;
  uint32 instance_number = 7;
  string fn = 8;
  string environment = 9;
  string etag = 10;
  uint32 lambda_type = 11;
  message ServerType {
    string kind = 1;
    string path = 2;
    uint32 port = 3;
    string protocol = 4;
    string method = 5;
  }
  message ClientType {
    string cron = 1;
    string headers = 2;
    string method = 3;
    string url = 4;
    string body = 5;
  }
  oneof type {
    ServerType server = 12;
    ClientType client = 13;
  }

  map<uint32, uint32> resources = 14;
  repeated string hosts = 15;
  repeated string dependencies = 16;
  repeated string arguments = 17;
}

```

Figure 50 ProtoBuf deploy lambda message definition

The “RemoveLambda” message has two fields: i) task id and ii) FQDN lambda name. The “DeployLambda”, Fig. 50, have all needed information to create/upgrade the sandbox containing the respective lambda.

```

message Event {
  Type type = 1;
  repeated string arguments = 2;

  enum Type {
    HTTP_REQUEST_EVENT = 0;
    HTTP_RESPONSE_EVENT = 1;
  }

  message HTTPRequestEvent {
    string path = 1;
    map<string, string> headers = 2;
    string body = 3;
    map<string, string> params = 4;
  }

  message HTTPResponseEvent {
    uint32 status = 1;
    map<string, string> headers = 2;
    string body = 3;
  }

  HTTPRequestEvent request = 3;
  HTTPResponseEvent response = 4;
}

message Invocation {
  string task_id = 1;
  string lambda_id = 2;
  string etag = 4;
  string project_name = 5;
  map<string, Event> events = 6;
}

```

Figure 51 ProtoBuf Invocation and Event message definition

Fig. 51, have all involved messages definitions in one lambda invocation. The point that we want to point out, is the events field. This field is used to chain different responses into one lambda execution flow.

```

message EventLog {
  string task_id = 1;
  string fqdn_name = 2;
  string node = 3;
  uint64 created_at = 4;
  map<string, string> event_log = 5;
}

```

Figure 52 ProtoBuf event log message definition

On single event log message can contain several messages, Fig. 52. Another important field is the created field. The created at field represents the Unix timestamp to keep the logs in temporal order even if they were received and stored out of order. The “fqdn name” already mentioned but never clarified, it is a URI scheme defined as:

*“fog://country\_code.node\_name.project\_name.lambda\_id.lambda\_version.instance\_number”*

At this point, the scheme only serves to physical localize a particular lambda instance within the network. Later on, can be used to provide new ways to invoke some lambda at a determined geographic place.

### 3.4.3 Runtime

The provider when starts the fog node, the node needs access to the bus. The connection string and the respective credentials are issued by the web platform. After successful authentication, Fig. 53, the node request to the platform to update the information about itself, this is because the provider could install a new environment or use another isolator, or new resources are available or just node upgrades or some failure. Another request that node does, is to update the protocol buffers file for the supported environments and the respective lambda API files, explained later on.

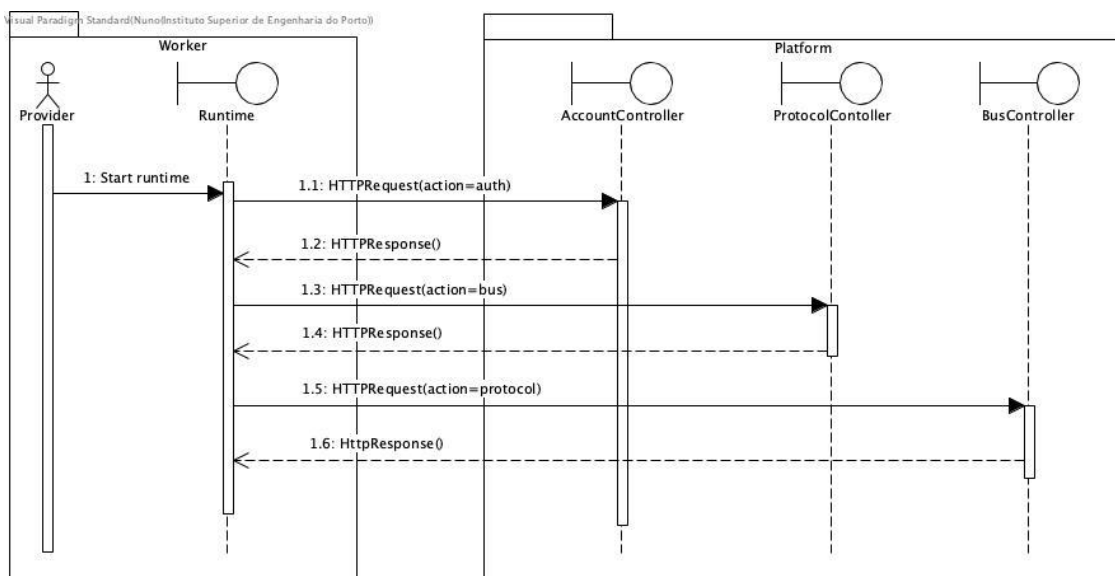


Figure 53 Runtime start up communications

Consider that the node had a power failure, and after a while, the node comes back online and the runtime starts again, in this scenario, the runtime will recover the previous state. Recover the state means that all the lambdas that were running before the power failure will be restored. This is done by the runtime maintaining local storage with the information about the system.

The deployment messages are delivered via the bus and the respective listener will call the right runtime service method. In the case of deploying a new lambda, the runtime service

(launchSandbox method) requests the isolator and then one sandbox is created for the respective lambda. The errors and other relevant information under sandbox creation process are sent via the bus to further processing. If the lambda even trigger is of type HTTP client, the runtime service will register the CRON job for that particular lambda and when fired, the sandbox will handle the time event by calling the lambda logic and sent the response via the bus to further graph execution by some scheduler.

When the client wants to remove/upgrade the lambda, the runtime service will call the graceful shutdown method, the lambda will be disconnected from the bus, and if the internal requests counter is 0, then the lambda will exit normally, otherwise the lambda will keep process the on-going invocation messages.

The runtime service at the start, it requests to cron service to fire a timed event to collect stats from the runtime itself and the deployed/run lambdas. The frequency is adjusted by the provider and can also be disabled, this is to reflect the device constrains.

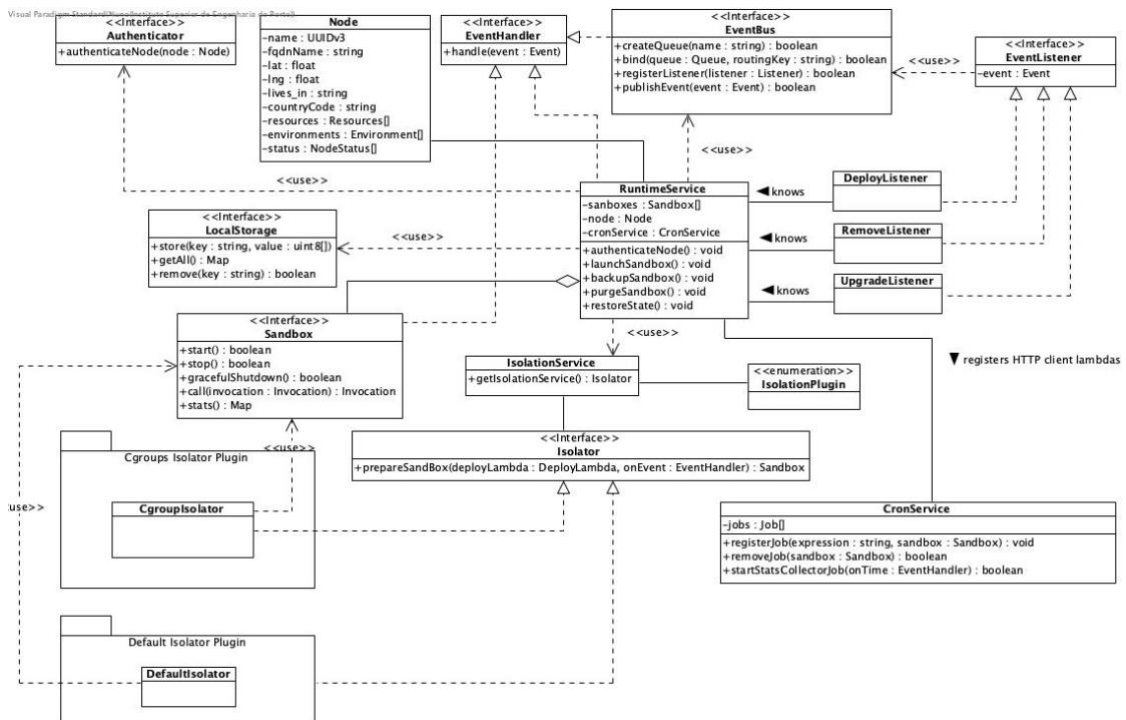


Figure 54 Runtime class diagram

The isolators implemented launch each sandbox on its process. The difference between the default isolator plugin and the control groups plugin is that the spawned process respects or not the resource usage limits defined by the client (Gonçalves, 2018). Both plugins deploy the

lambda under a well-defined directory structure under some directory specified by the provider. For example, a specific lambda can be found under the directory:

*“file:///provider\_defined\_dir/project\_name/lambda\_id/lambda\_version”.*

The isolator creates the following files and folders under the lambda directory: i) dependencies folder, ii) lambda file, iii) protocol buffers file, iv) entry file and v) the lambda API file. The first will contain all runtime dependencies, client defined dependencies plus system dependencies, that the lambda needs during its execution. The lambda file is the one that contains the logic written by the client. The protocol buffers file is the file that is generated from the messages definitions by the protocol compiler targeting a specific runtime environment(s). Some variation is expected into the defined protocol, affecting this way the generated files which can lead to redistributing the resulting protoc files to all fog nodes. The protocol buffers tool guarantees back compatibility yet make it the most up to date is a reasonable aspect to take into consideration, with this, resides the need at start, Fig 53, to update, if needed (conditional request), all protocol files from each different supported environment.

The next file is the entry point. This file has the logic of establish bus connection, using the connection string from the process argument, register the invocation listener, insatiate the lambda API class per invocation and publish stats to the bus.

The lambda API file is one file that is responsible to call the lambda logic. The file injects as function argument the context object, Fig 40. The context object has a set of helpful methods to unpack the invocation message, Fig. 55.

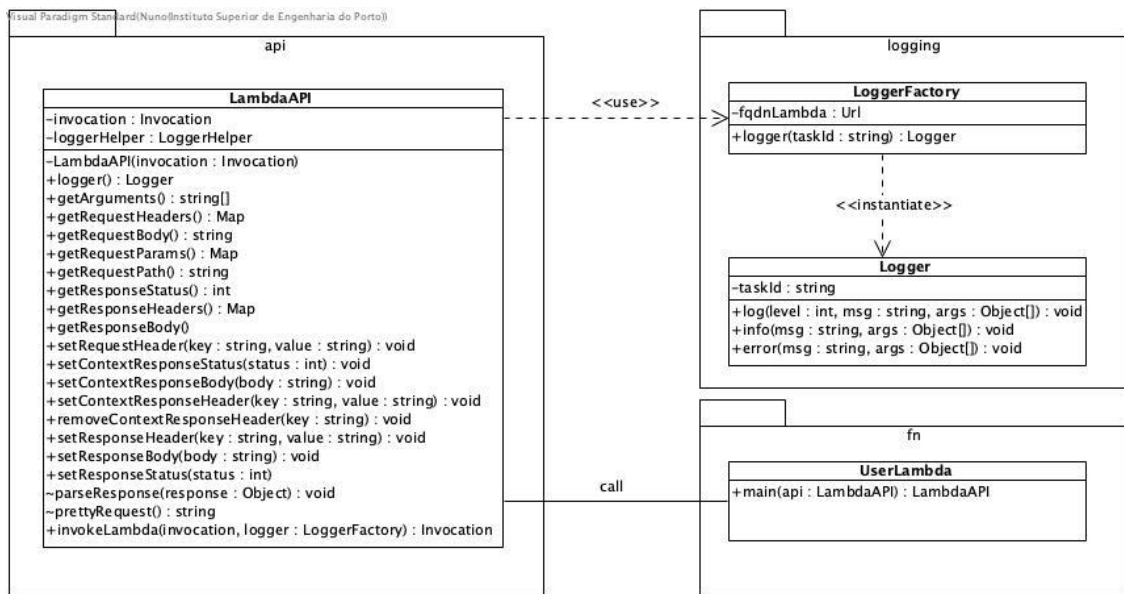


Figure 55 Lambda runtime API

We conclude the design and the implementation section. We have discussed all important designs and their implementations. In brief, the client creates logic execution units with the help of the web platform, the platform and the scheduler process the deployment and the consumer can start making requests. The request triggers one lambda graph execution, which can lead to the execution of innumerable logic units at different nodes.

### 3.4.4 Alternative Design & Implementations

In the state of art chapter, in the research part of answering the “where” question, various designs were discussed, and all are possible architecture alternatives to implement this paradigm. We have identified the following plausible alternative designs/models that are related to our solution: i) platform as a service and ii) infrastructure as a service.

The hardware virtualization could be done by using Apache Mesos, where is a platform for sharing commodity clusters in a fine-grained manner (Hindman, et al., 2011). To have infrastructure as a service, we must be capable of launching virtual machines under the cluster, for that, we could use Vagrant (HashiCorp, 2019). Vagrant uses a declarative syntax to build and launch portable virtual environments and supports a wide range of virtualization technologies like Docker, kernel-based virtual machines, VirtualBox, Hyper-V, and AWS, etc. Platform as a service could be implemented using Apache Marathon (Apache Marathon, 2019), where is exported a REST API to the developers can send their bundled application.



Both solutions require hardware virtualization, consequently all fog nodes must have reasonable computational resources to handle virtualization technology, this fact makes smartphones disqualified to be a fog node due to the complexity of installing and set up this kind of systems, at this time, we do not know any kind of solution on Android platforms or other mobile platforms that make this possible, while we offer customizable isolation, low size runtime binaries with easy installation steps, even for Android mobile phones. The solution allows us to make our smartphone into our personal webserver if we need to.

Another disadvantage is that the developers must deploy their applications at one or more specific pre-defined physical places, this is, they must select and deploy their application over an available place, making the offloading from the fog to the cloud and vice-versa more rigid and less flexible when compared to our proposed solution, since we can select multiple nodes per lambda, and those nodes can be fog and/or cloud nodes. One advantage of the alternatives is the flexibility of deploying other types of applications, that work with different network protocols.

To conclude, our solution allows easy integration of computational power into the network. The function as a service empowers fog-cloud interplay in a fine-grained manner, offering to the developer a powerful tool with a familiar protocol (HTTP) to design and create new types of applications. The alternatives not allowed this seamless fog-cloud interplay, pushing this responsibility at the design phase of the application with strong constrains in the future development of the overall application.

## 4 Results and Discussion

This section has the procedures that we have conducted during the test stages. The developed platform has designed to handle with a wide range of applications however, we reduce the application surface into a REST application and a movement detector over a real-time stream of video.

The test of the REST API application scenario resides on the fact that this architectural style is predominant in the industry, with this, we can prove the functionality provided by the platform, while the second application, is an application that benefits from the paradigm shift due the operational cost for large-scale deployments.

### 4.1.1 TODO Application

#### 4.1.1.1 Objective

We prepared this test with the goal of measuring the throughput of the lambda(s), plus to follow the CPU, memory and bandwidth usage across multiple deployment scenarios, more specifically, how those metrics behave when comparing the platform against a standalone web application in a local environment and on a standalone web application in a cloud environment with different number of workers.

#### 4.1.1.2 Environment & Configuration

It was set up two applications under two different deployment scenarios. One application was developed under the platform using NodeJS as the environment. The other application, is a

standalone application, also constructed in NodeJS, with the same framework (Express) that was build the scheduler. Both applications export an interface to manage TODO's, Fig. 56. There are two different actions, one is to retrieve all entities while the other is to persist a new one, in both cases, the representation is in JSON. The entities are stored in the filesystem as clear text in a CSV format.

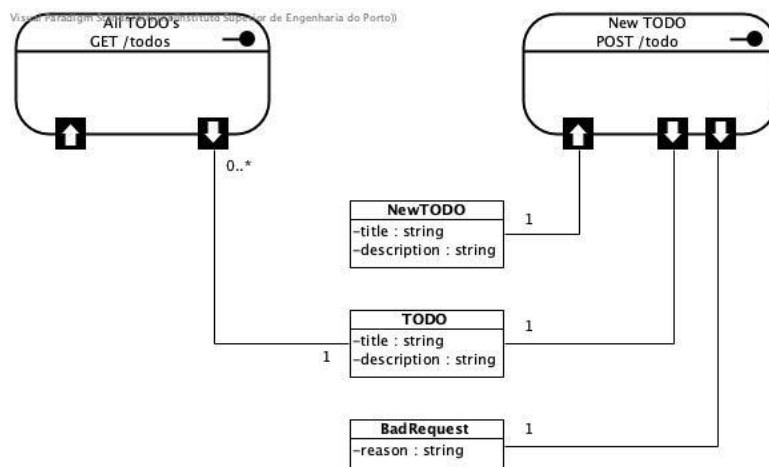


Figure 56 TODO REST API

The first deployment test scenario is a local deployment, where all applications are hosted in the same machine, Fig 56. In the second test deployment scenario, the standalone application was placed at the cloud while the platform stack was deployed at the fog. The fog is 2-hops way from the "local-node" machine.

The test consists of 6K concurrent HTTP GET requests, where only one TODO entity is retrieved. For each deployment scenario, the benchmark was executed with a different number of workers. In the first round is used one instance of each application, the second, 3 instances of each and finally, the third with 6 instances of each. To get multiple instances on the standalone application a load balancer was deployed in front of the workers, in the platform stack, the load balancing is already built-in feature.

Only one entity is retrieved because we are focused on the throughput of the two applications, and the size of the payload must remain as little as possible to fit in a single PDU, avoiding IP fragmentation. The HTTP uses the TCP/IP stack, the TCP protocol adds 20 bytes and the IP adds another 20 bytes while the ethernet uses 14 bytes, summing up, 54 bytes are used just for those protocols, assuming the worst scenario of an MTU of 512bytes it leaves of 458 bytes for the HTTP protocol.

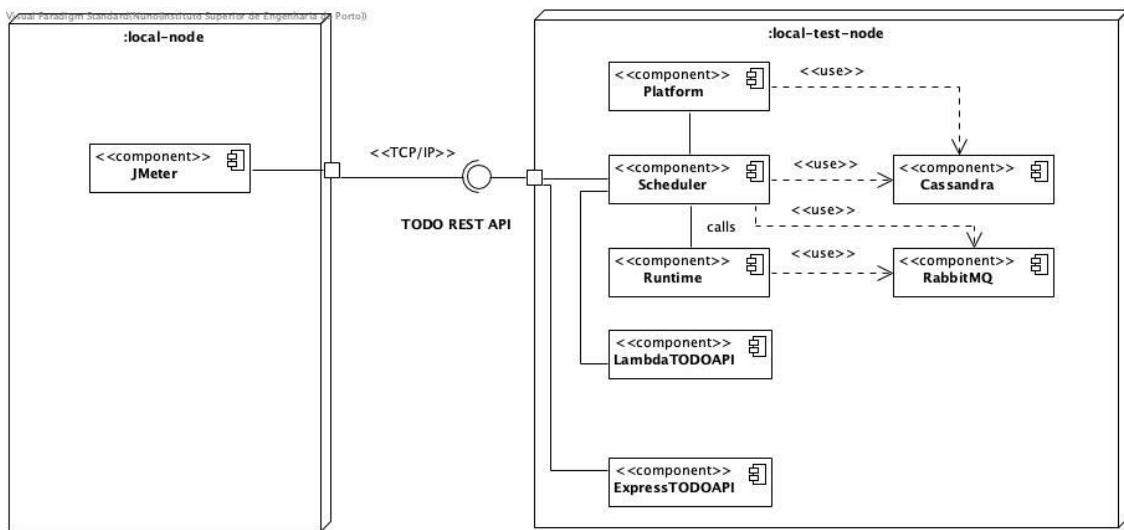


Figure 57 Deployment scenario 1: local deployment test

In terms of node specs, the node “:local-test-node” and the “:fog-test-node” has 16GB of RAM and an 8<sup>th</sup> generation Intel Core i5 that has 6 physical cores with a base clock of 2.8GHz and a maximum of 4Ghz. The “cloud-test-node” is hosted by the Google Cloud Platform in Iowa, USA datacenter, and is a custom instance with one dedicated virtual CPU and 4GB of RAM, the CPU is an Intel Xeon with a clock of 2.30GHz.

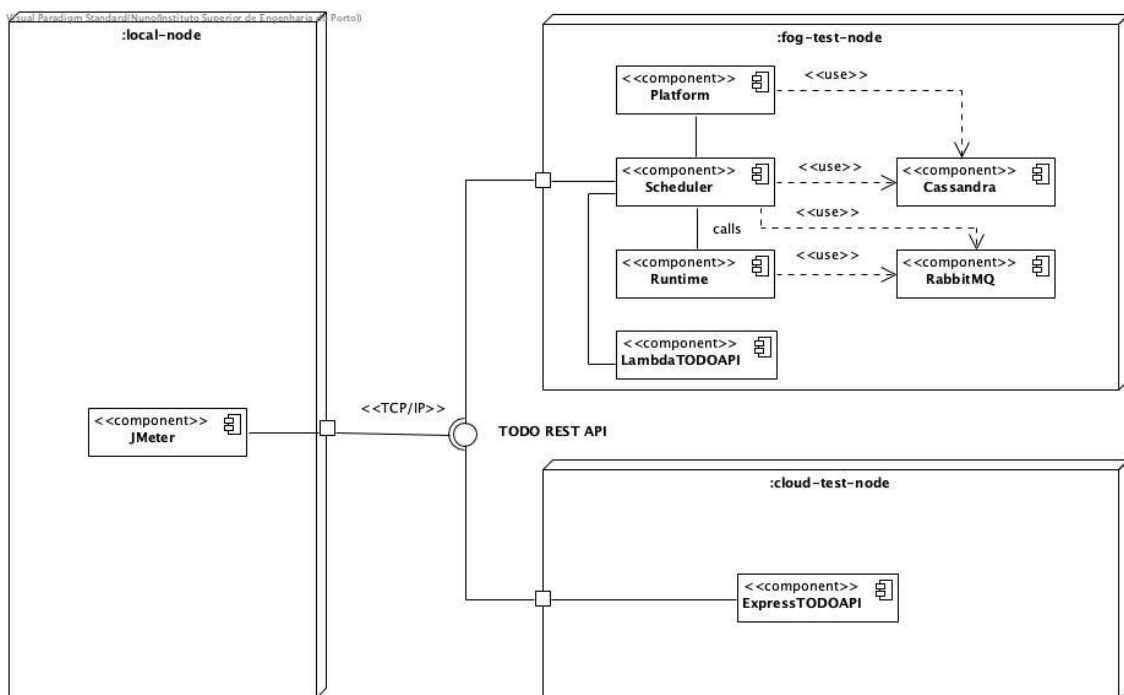


Figure 58 Deployment scenario 2: fog and cloud deployment test

The tool used in both deployments to launch concurrent requests was the Apache JMeter. We have defined 12 threads to launch the already specified number of requests for each round with a ramp-up of 0s. The JMeter HTTP client also was set to have a 2s timeout and 2 listeners were added: i) assertion result and an ii) view result tree.

The application developed in the platform can be translated by the creation of one project, with two lambdas: i) create-todo, and ii) list-todo. The create-todo lambda has the logic of insert entries into the file, while the list-todo lambda retrieves those entries from the file. The runtime did not apply any kind of process isolation, in this test, the lambdas could use the resources that they need without supervision.

#### 4.1.1.3 Results & Discussion

None of the tests had failed requests, with this, each round transferred a total number of 1668000 bytes, being 426000 bytes correspondent of the body content (71 bytes per request) and the remaining is respective to the HTTP protocol (207 bytes per request). This numbers also say that each response from the lambda and the standalone application have the same body and the same headers, with a final PDU size of 332 bytes.

Not having HTTP request failures for the lambda TODO app, also means that the underline development is working correctly because when a socket is open between JMeter and the lambda scheduler, the socket remains open until a 2 seconds JMeter client timeout (it was considered as an error) or response is given. The developed platform uses high-level APIs to respond to the clients and it is not possible to shuffle responses with different sockets. The only possible response is the correct one, plus, we have added the assertion result listener to validate the content of the response.

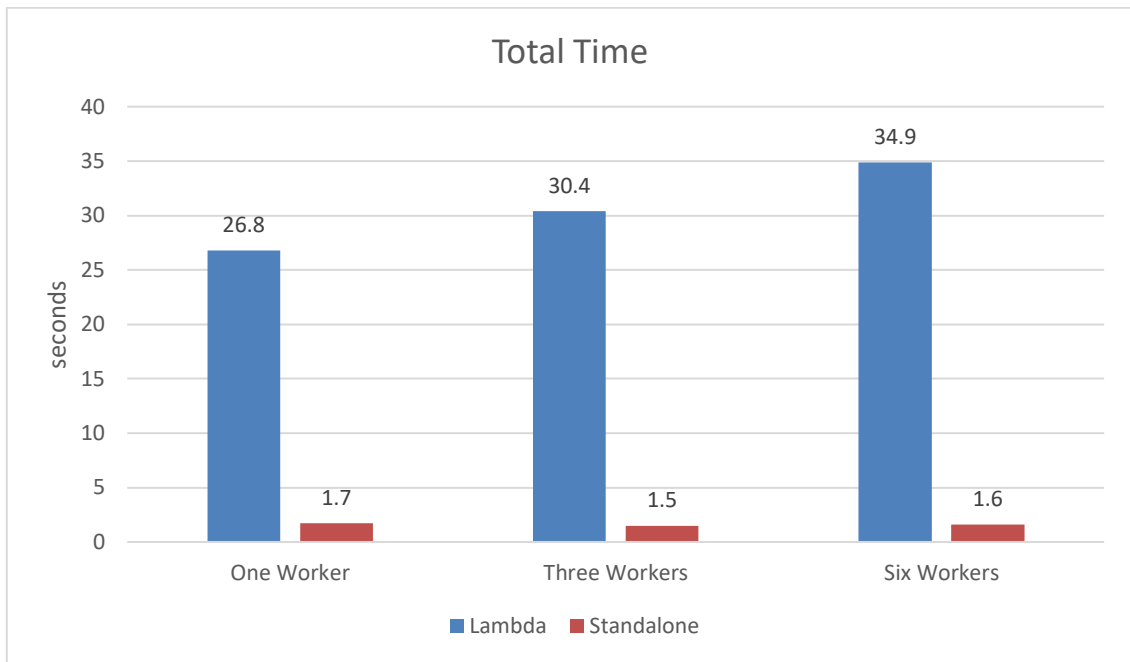


Figure 59 Local deployment time spent

The standalone application in a local deployment is almost 19x faster than the lambda, Fig 59. The variation of the number of workers is irrelevant for the standalone application, while for the lambda the increasing number of workers had the inverse effect.

The fact of the lambda app being slower was expected since the communication between the JMeter and the application is a direct connection via one socket, in counterpart, the lambda TODO app receives the request via the bus and publishes the response into it, and finally, the respective scheduler forwards the response back to the client.

The guarantee of the at-least-once message increases latency in the bus, in other words, each request takes a mean of 53ms (one worker test scenario), 51ms is spent on the bus, while the standalone TODO app only has a mean of 4ms per request.

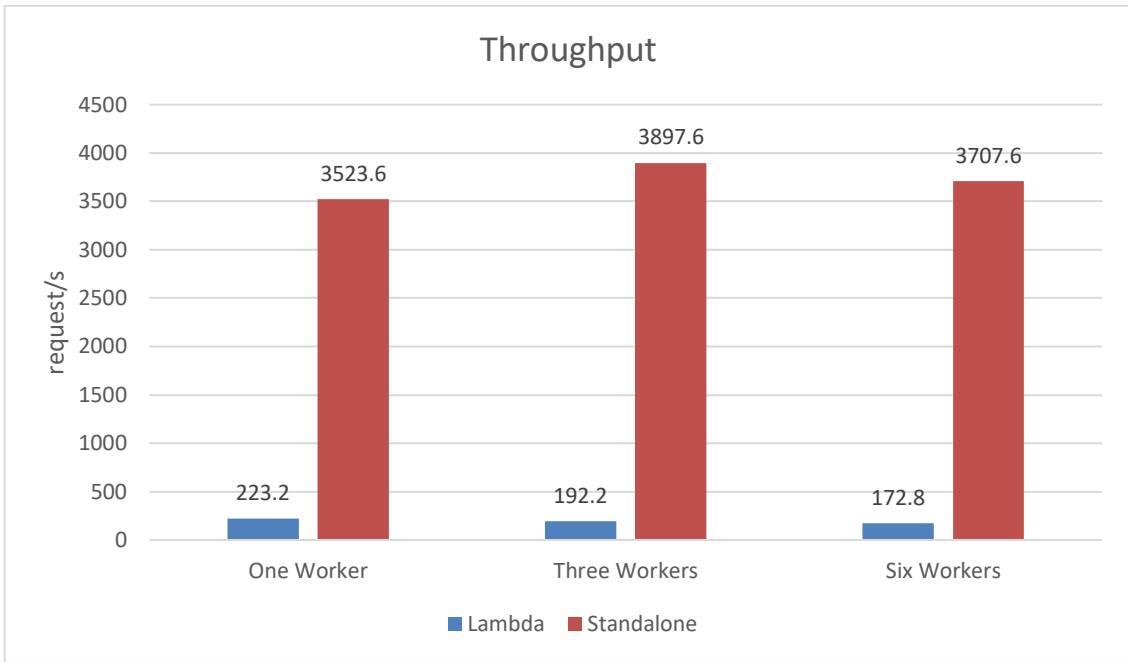


Figure 60 Local deployment throughput test

When we increase the number of workers of the TODO lambda app the throughput decreases. This occurs by the fact of each message publish requires 2 acknowledgments, one is for the exchange to the publisher, and another is for the exchange to the consumer, plus, each queue in the RabbitMQ is managed by one erlang thread, which under stress the amount of traffic affects overall throughput.

The CPU and memory usage represented in the following figures, for three workers and six workers is the mean of the respective CPU/Memory usage of the workers for each round. The CPU usage benefits from having more workers, the number of non-voluntary context process switching decrease 3x when used three workers and 18x when using 6 workers when compared with the one worker solution.

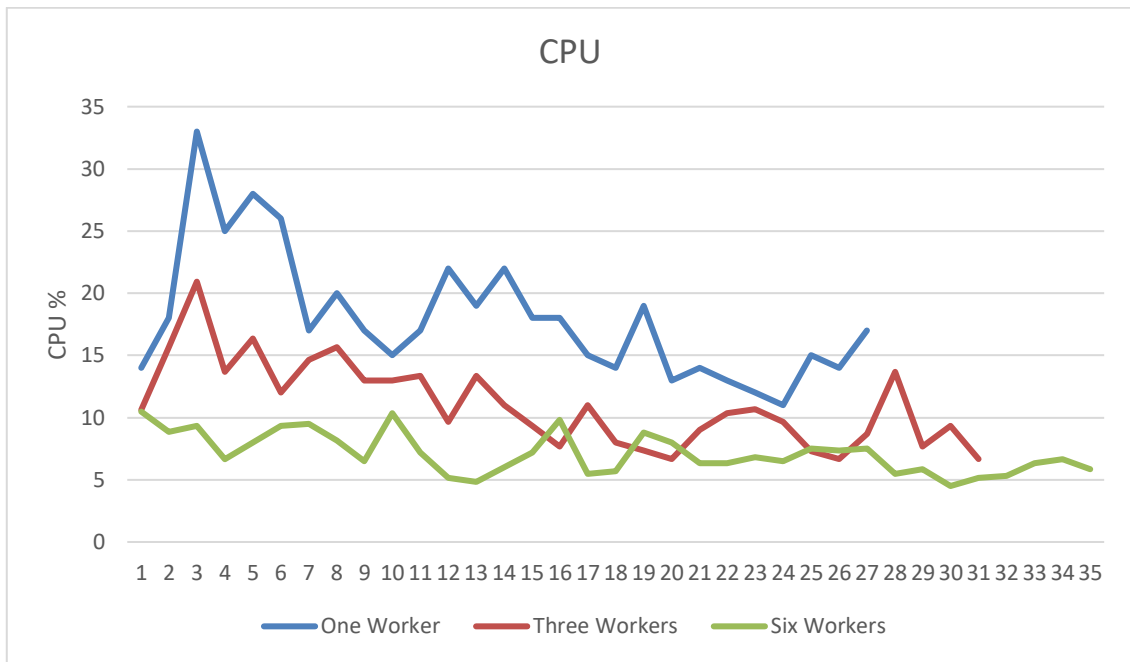


Figure 61 Local deployment lambda TODO app CPU usage

The standalone uses all CPU power, Fig. 62, to deliver the responses when only in worker is being used. With more workers the CPU no longer hits its peak and observes fewer number of non-voluntary process context switches.

The memory numbers illustrated is the resident set size RSS, which is the memory that the process in the RAM. It includes all stack and heap process memory and also the memory of shared libraries as long as the pages from those libraries are actually in the memory.



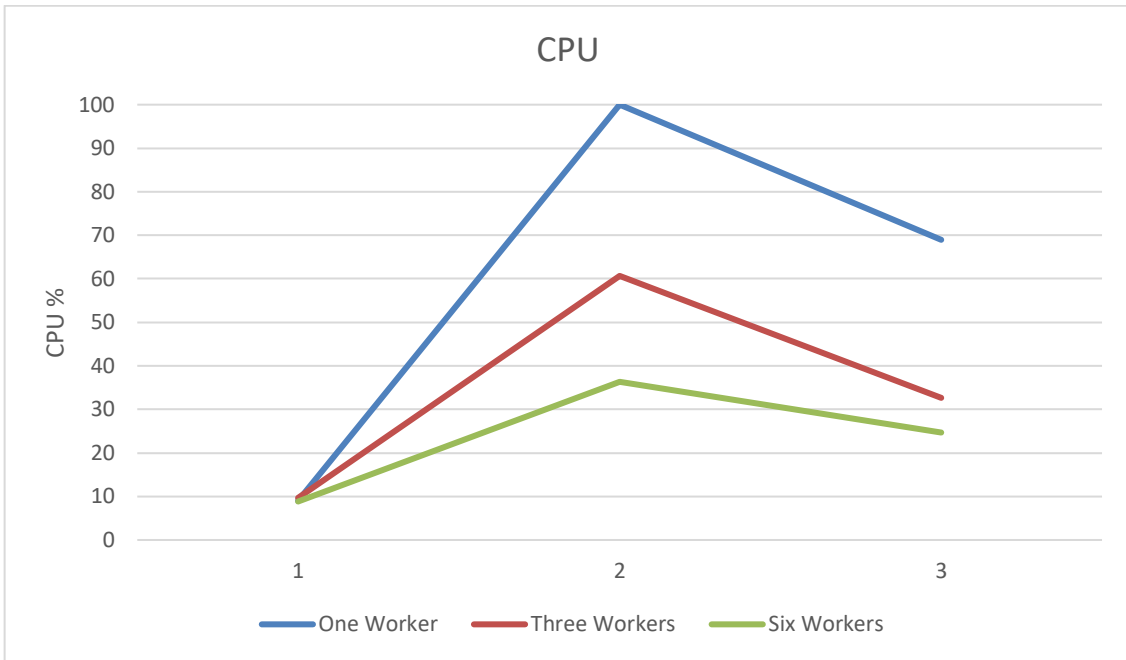


Figure 62 Local deployment standalone app CPU usage

A NodeJS process in an idle state consumes a mean of 40MB of memory RAM. The number of workers decreases the number of memory RAM used relatively, however, when using 3 workers and 4 6 workers we have to multiply by 40MB by 3 and 6 respectively.

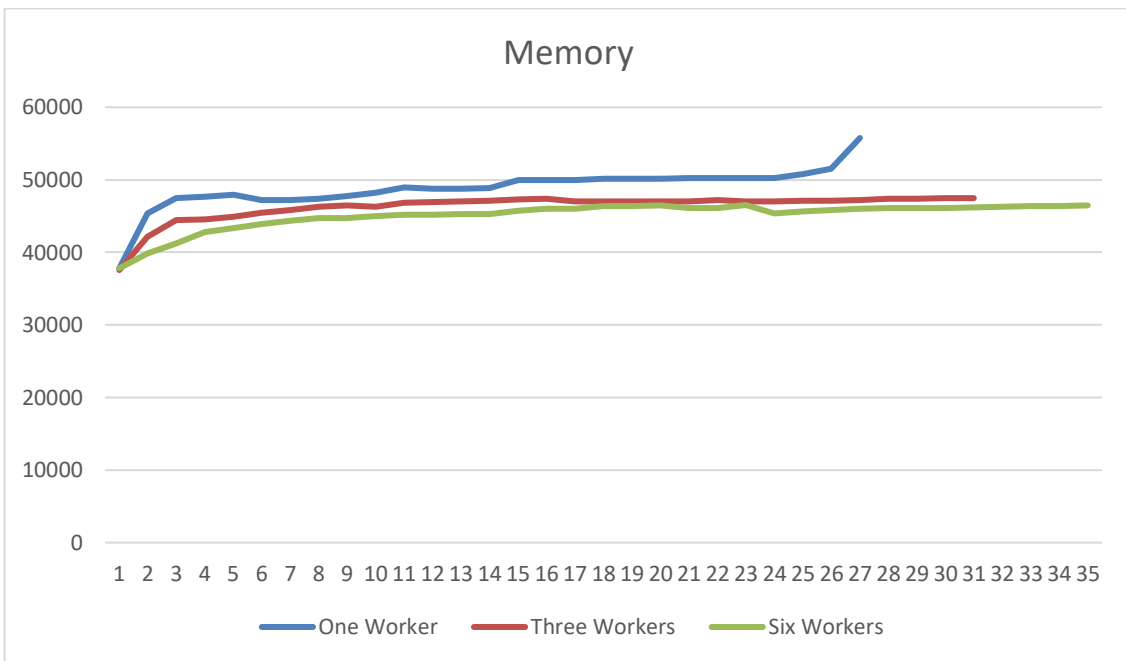


Figure 63 Local deployment lambda TODO app memory usage

The standalone application shares similar memory usage when compared with the lambda TODO app across the different number of workers. The TODO retrieval entity is not particularly memory intensive and this is reflected in the memory used by both deployment scenarios on different tests.



Figure 64 Local deployment standalone app memory usage

The standalone application was closer 4x slower completing the 6K HTTP request tests against the fog lambda deployment, but the more impressive metric is that it was 84x slower when compared with the local deployment. The lambda TODO app on the fog deployment added a mean of 12s of latency when comparing to the local deployment.

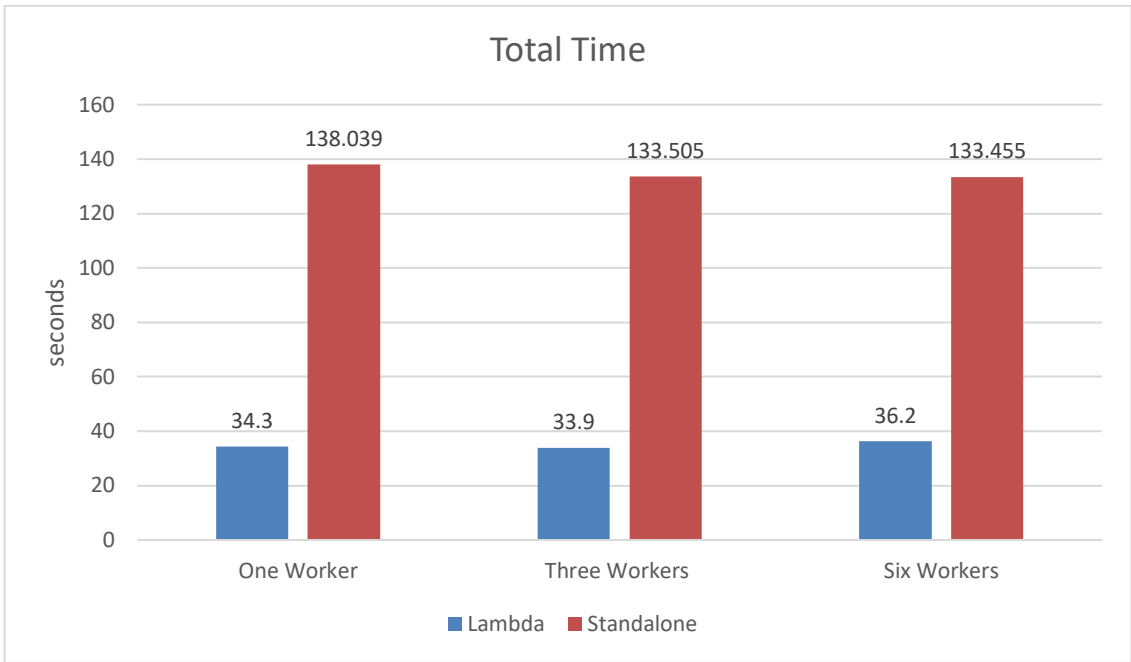


Figure 65 Fog-cloud deployment time spent

The fog-cloud served a mean of 172 requests per second while the standalone had served 44 requests per second, another perspective is that the lambda had served more 128 requests per second. The number of workers has no impact on the standalone while the lambda had a small increase when moving to tree workers, however, it decreased when six workers were added.

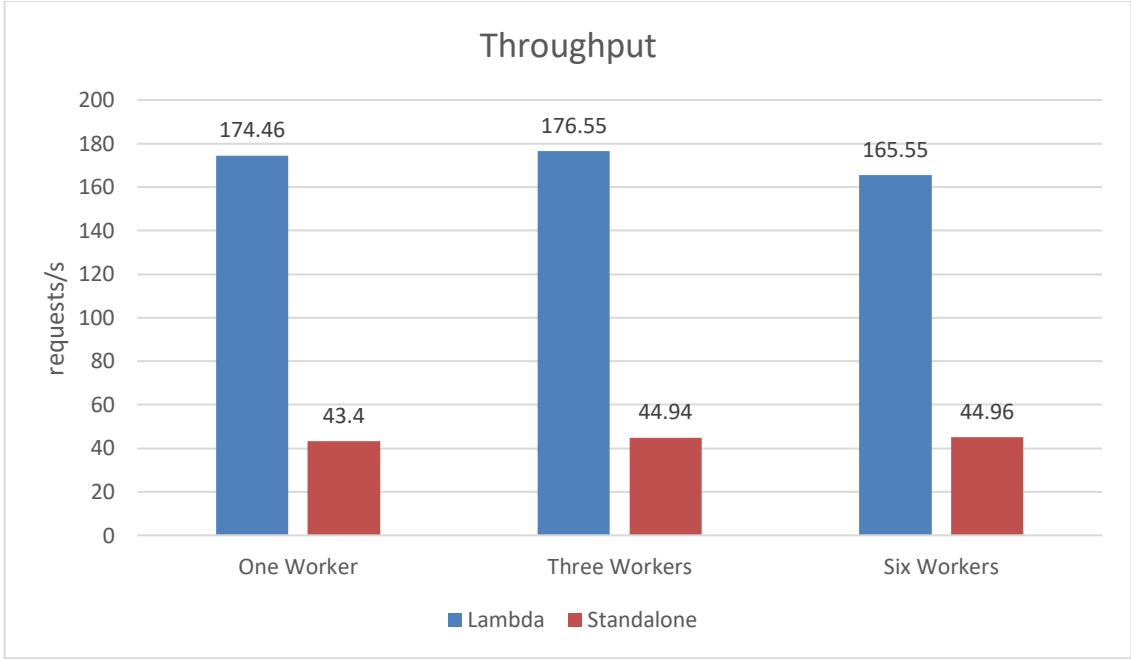


Figure 66 Fog-cloud deployment throughput test

With more workers, more requests are distributed, and the mean of the CPU usage decreases. Compared with the local deployment the mean difference is 1% when one worker is used, 4% when three workers and 1% for six workers.

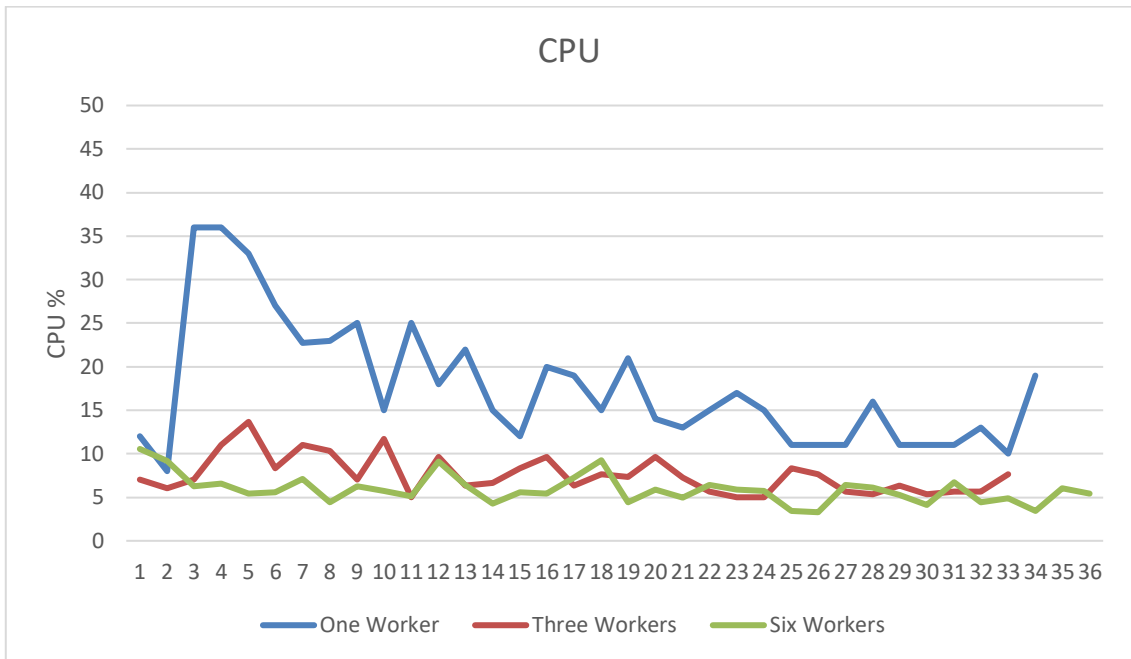


Figure 67 Fog-cloud deployment lambda TODO app CPU usage

One significant difference is in CPU usage between the local deployment and cloud deployment for the standalone application, Fig 68. The mean difference for one worker is 56%, 35% for three workers and 21% when six works, for the CPU usage.

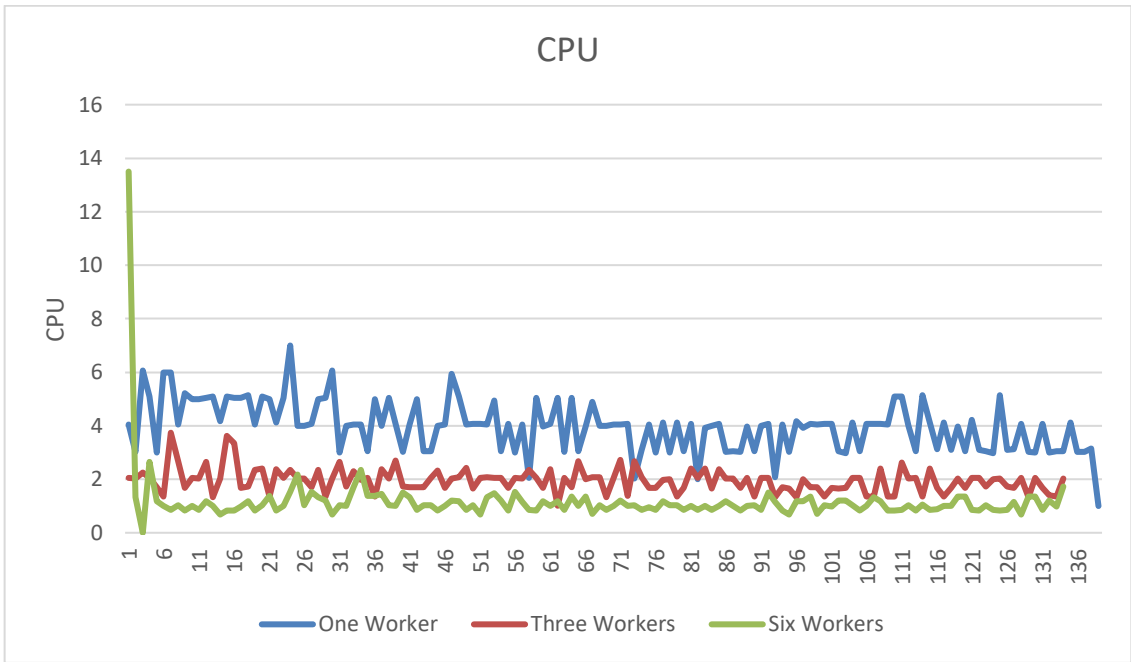


Figure 68 Fog-cloud deployment standalone app CPU usage

The memory has 1.5MB of deviation between the local deployment and the fog deployment for the lambda TODO app, across the different number of workers. This number also shows that memory it behaves similarly in both deployment scenarios.

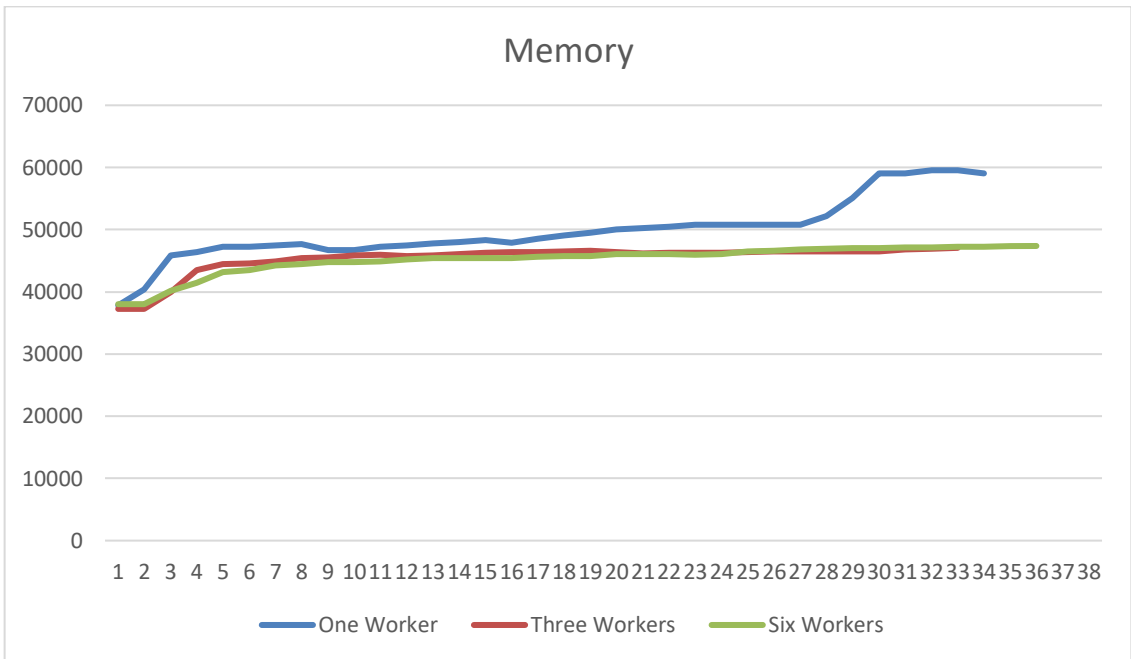


Figure 69 Fog-cloud deployment lambda TODO app memory usage

The standalone application in the cloud the memory usage for three and six workers followed the mean of an idle NodeJS process, 40MB, and consume less 4MB on average with their respective in the local deployment. One worker stays with a mean of 42MB, less 6MB when compared with the local deployment.



Figure 70 Fog-cloud deployment standalone app memory usage

The tests executed reinforce one important aspect that was being discussed on this entire work: the WAN latency. Focusing only on the standalone application and the latency between the two deployment scenarios, on one hand we have one response within a mean of 3ms and on another hand, we have one response with a mean of 266ms. If we consider the 3ms pure process execution since we are in the local deployment scenario, we have 263ms that are lost in the wire.

One web application will always be deployed in some datacenter infrastructure of some company or cloud provider, and in this particular case, it is 84x more important the physical deployment than the resource that the host machine has. The figures of the CPU usage in the different scenarios (local and cloud), illustrate perfectly the previous statement. The CPU on the cloud barely have used during the test.

Response times in the order of 266ms or even 2.5s is reasonable in today's web applications (Littledata, 2019), however for other applications such as autonomous driving (sensors), virtual

reality, video analytics (autonomous driving with 30FPS), tele-surgery where latency is expected to be 10ms, 15-7ms, 50ms and 150ms respectively (Lema, et al., 2017).

The tests show that the longest lambda TODO app request is 151ms over with a maximum mean of 72ms per request. These numbers are from the same scenario which is: fog deployment with 6 workers. Considering always the worst case, the developed platform is not well suited for ultra-latency applications, although we experience responses within 4ms. Achieving constant 4s responses in two-hops away application was what we have tried without succeeded after results from those tests. We have changed how the RPC is made in two different forms, using one queue per response and use the same queue for receiving all response, on the message itself we have changed from durable to transient messages, on the connection side, using dedicated channels from receiving and publish. Note that the property of at-least-one message was always preserved during the modifications, worse than high latency is no response at all, and yet, none of the mentioned approaches change the latency experienced in the bus, which for ultra-latency application we can conclude that with current technology, RabbitMQ, cannot be used with that purpose.

The platform is still a value vehicle to deploy REST applications using all kinds of devices to serve the requests. We have demonstrated that the developed architecture can reliably respond to the clients with a moderate latency requirement. The list TODO lambda sandbox uses 2.4MB of the disk, with an average of 40MB of memory RAM usage per process.

#### **4.1.2 Motion Detection on Real-Time Video Streaming**

##### **4.1.2.1 Objective**

The application has the functionality of detecting movement over a stream of video, intending to demonstrate the lag between download, processing and trigger an action, when motion is detected in fog and cloud deployment scenarios. The application can be seen as a starting block for a greater image processing pipeline, for example, in a traffic monitorization context, once movement is detected, one video segment is sent to a heavy, cloud hosted, DNN model for object detection succeeded by a car crash classifier. The IHS in 2015 states that in 2015 we had 1 camera installed for every 29 persons on this planet, and the Microsoft researchers state that video analytics is the killing app of this new paradigm model (Ananthanarayanan, et al., 2017).

#### 4.1.2.2 Environment & Configuration

To detect motion in the video, it was used the background subtraction technique, Fig. 71, where is used a static background image and from this background is subtracted the current frame, leaving with changes of the current scene. This technique is used with stationary cameras and is widely used for video surveillance, traffic monitoring, etc. We have used OpenCV (OpenCV, 2019), a library for computer vision, the results are illustrated in Fig. 70, where we applied the subtraction (frame in the bottom left), plus a contour recognition. The contours are obtained using the mask and applied, just for visualization, to the original frame (green boxes). In the top right of the picture we have two numbers, the first is the number of the frame that is being displayed, and the number below is the number of objects that are being recognized. The program that analyses the video stream outputs the median of the number of objects recognized in every 30 frames or 1s.

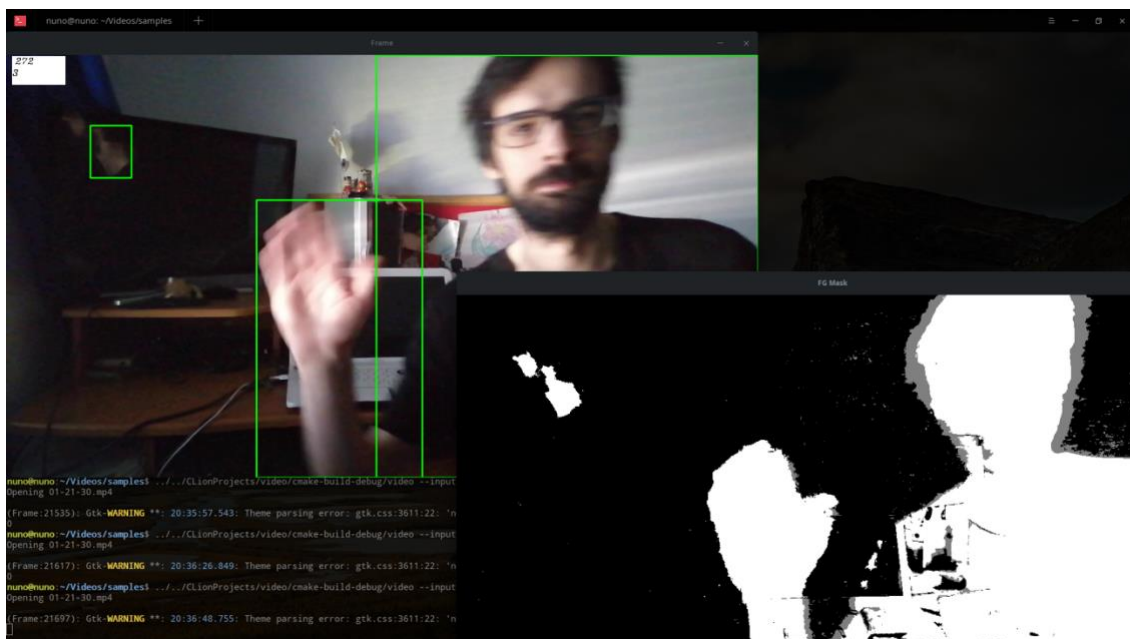


Figure 71 Background subtraction

The video was recorded with 3minutes (m) with one human motion starting at 1m and 20s up to 1m 35s. The video was recorded with a resolution of 720p at 30 frames per second. The video was sliced into small clips of 10s each with 12MB in size, MP4 encoded, making 18 clips in total. The clips then where moved to a mobile phone and they were served via an HTTP server. The video consumers were placed at fog and cloud where the metrics were collected.



The server in mobile phone exports a single URL that in every ten seconds makes available the next clip, with this, it was like the phone was capturing the video and created the encoded clip for every 10s. This scenario allows that the video analyzers process the video at their own pace, otherwise, in a multiple producer scenario, the server could be overloaded with data slowing down the process. The consumer defines an HTTP header that identifies itself, so the server in the mobile device keeps track of the last downloaded video by that particular consumer. The cloud and fog nodes are the same that were used in the previous test application. The mobile phone used a wireless connection to connect to the ISP router and is 2-hops away from the fog server. The platform hosted all code involved in this application, including in the smartphone since we have NodeJS pre-compiled for ARM processors. To program that is deployed in the cloud and on the fog makes a clear distinction of two components involved in the video analytics application: i) download the video; ii) processing the video and iii) program logic. The video processing only starts when the video clip is downloaded from the mobile phone.

4.1.2.3 Results & Discussion

Analyzing 18 clips of encoded videos with 10s each, took 4.7m on the fog, 1.7x faster than the cloud, where the same test took 7.9m. In the fog scenario, the network expresses 10.8% of the total execution time and the rest is to video analysis, in counterpart, the cloud takes 37.5% and 63.3% for network and video analysis respectively.

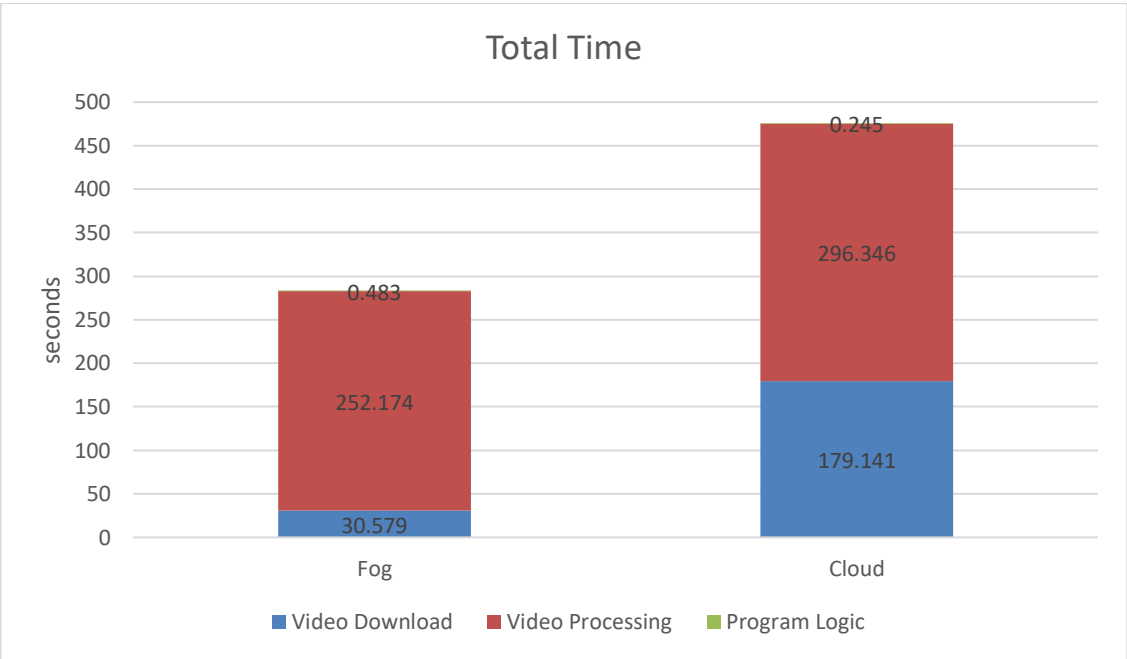


Figure 72 Video application execution time

The link of the mobile server has an Internet upload speed of 11MB/s, the total bytes downloaded by the two applications was 216289807bytes, where the fog has downloaded on average of 7MB/s and the cloud at 1.2MB/s. The cloud deployment takes an average of 10s, with an amplitude of 1s, to just download one clip of 10s. Another perspective on the current scenario, is that the clip, which is already encoded to save bandwidth, with a 10s lag between his recording and to be ready to be downloaded, the cloud adds another 10s, which if the video was processed instantaneous, the trigger to answer to the eventual moving detection, at worst scenario (movement in the last second), occurred 20s later, while the fog could trigger the same action in 11.7s.

The video is not processed instantaneous, the background subtraction takes time, and this time constrains the rate of processed clips. In this discussion, since we do not have the same CPU in the fog and the cloud scenario, but we know how much time it took to video processing under both deployment scenarios, we have considered the time that was taken on the cloud environment. The reason behind this consideration resides on the fact of the fog CPU is targeted to end consumers in contrast to the cloud, that has a Xeon, a CPU aimed for server's workloads, with energy efficiency in mind. The number in the further discussion is always the worst-case scenarios, this is, the maximum time to download the video on each deployment scenario and the maximum time that took to process the video, considering only the cloud server times due to the reason mentioned before.

The 18 clips that were processed to detect motion, each clip under the CPU takes an average of 16.5s, with the fastest time of 16s and slowest of 19.2s. Considering the cloud deployment scenario that in the worst case, a 20 lag in transit (record and download), plus a 19.2s in video processing, results in 39.2s of total lag, moreover, in sequential execution, in each clip analysis, the process accumulates 2 clips, in transit accumulates 1 clip and by the time that completes one cycle, 3 clips are already ready to be downloaded, and when downloads the next and start to process the clip, the total lag could be of 50s.

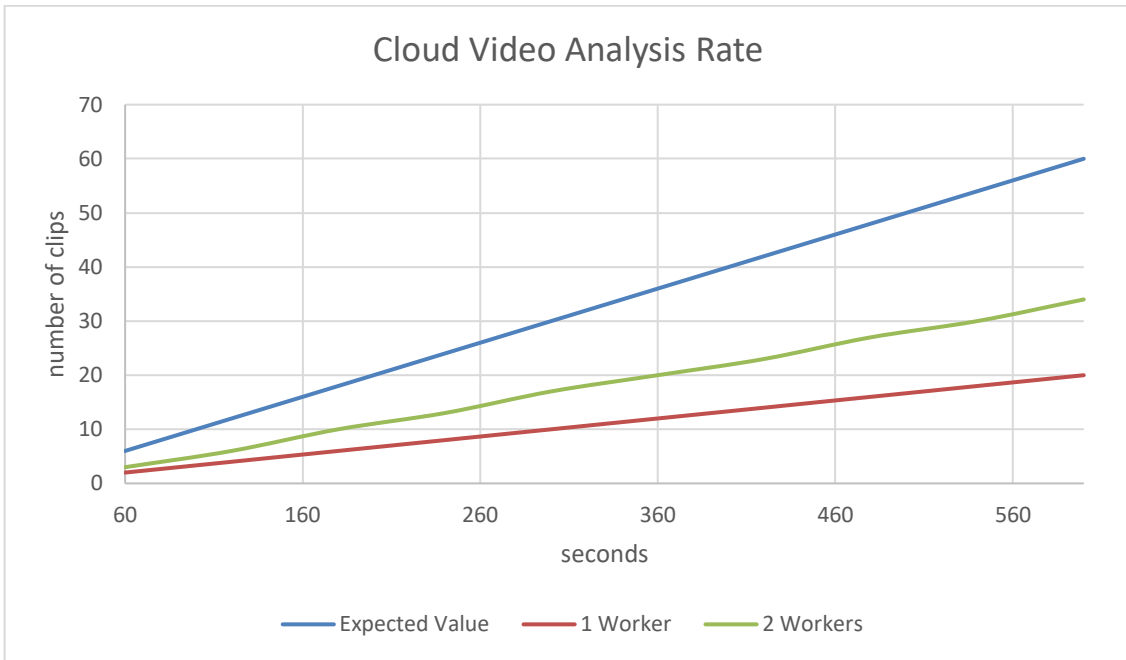


Figure 73 Cloud video analysis rate

To keep the application with the shortest lag, 3 workers would be needed, Fig .72. Figure 72 was removed from the first special case of  $t_0$  where no video is available, and the worker(s) would be put in the waiting list until  $t_1$ . With the current cloud deployment, after 10m, the application had processed 20 of a total of 60 clips accumulating 40 clips or with a time lag of 6.6m.

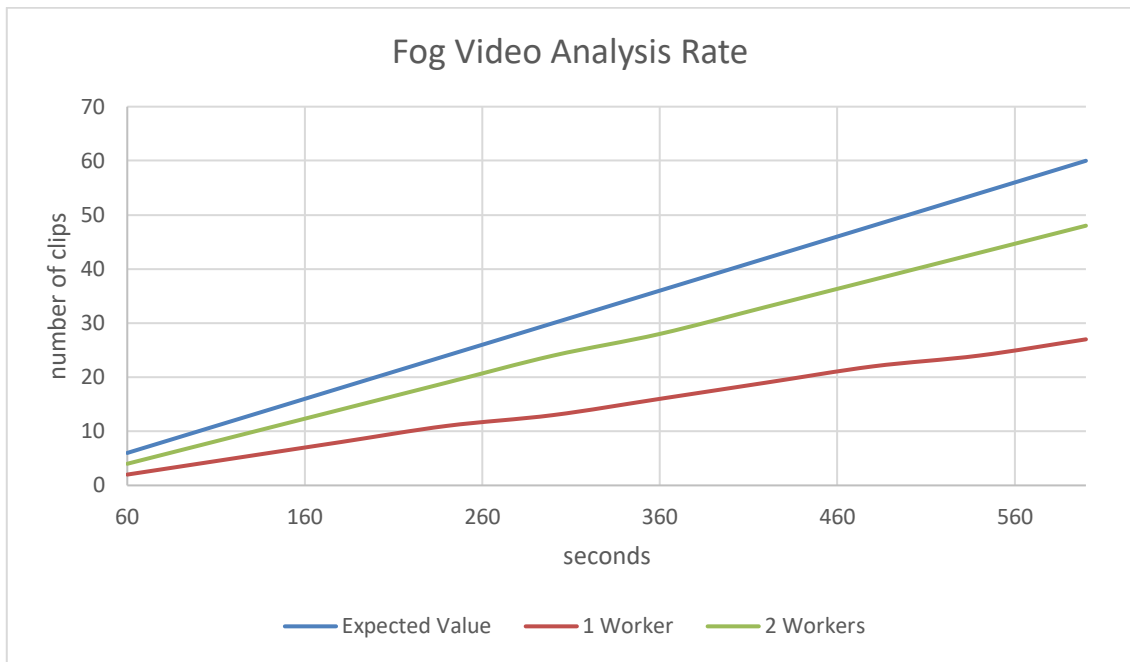


Figure 74 Fog video analysis rate

The fog deployment with one worker in 10m accumulates 33 clips or 5.5m of time lag, 1.1m less than the cloud deployment. However, to keep with the shortest lag, it also requires 3 workers or 2 if the CPU processes each video within a 17.5s window. Take the time to within 17.5s is reasonable to get there, plus of the 18 CPU records, only two are greater than 17s, since we are taking always the worst case, we estimate 3 workers to the fog deployment, this to point out, for the cloud, reducing two 2 workers, will need a video processing within 10s, which only with advanced code optimization, such using CPU SIMD instructions or GPU or even dedicated hardware, to get this number lower than 10s.

In terms of bandwidth for cloud scenario, serving one clip over the current link is in his best possible case because 1s is equivalent of 1.2MB and the real upload speed was 1.2MB/s. In the scenario of 10 cameras and if the bandwidth was equality divided between the cameras, we could send 3 frames per second, each clip would add a lag of 100s.

According to (IBM, 2019), for video analytics we need a frame rate of 12fps, Bosh (Bosh, 2016) suggest 15fps, so dropping frames to only send 3 per second is mentioned as non-practical, although, everything dependents under the business context. The best ratio under the current link is to use 2 cameras, recording at 15fps. On top of all resource constraints, we have the cost per byte.

Geographically distributed cameras are usually arranged into clusters and each cluster uses some sort of mobile or wired connection to stream the data (Ananthanarayanan, et al., 2017) to some datacenter. Taking our application and considering that the smartphone use the 4G for downstream the 3m of video or 216MB, according to the average cost of 12.77€ in moving 1GB, using mobile connection (Cable, 2019), we will have the cost of 2.8€, if we streamed 24 hours we would have a total cost of 1323€.

Before concluding, in terms of relevance of the developed platform in this application, and for this paradigm shift, is on the deployment simplification, workers management and monitoring during the execution of the various trials and the final test that were performed. The real value comes in the form of cost reduction in video analytics, reducing bandwidth and consequently processing video faster and actions can be synthesized faster improving the overall system. Due time constrains, we do not design an application that would integrate the two environments, fog and cloud, where clips could be forward to the cloud when its capacity reach some limit or vice-versa, however, our point is that the developed platform merges the two environments to implementing this interactions due the node location awareness and an unified management and control.

To conclude, this application captures the requirements for large-scale video analysis. The bandwidth is the one variable that we cannot change in the same sense that we easily process the video on more powerful CPU, or developed more optimized code to reduce processing time, the same options are not available for the WAN, the current infrastructure is not even prepared to move a huge amount of data. The latency of the current application could be easily improved by instead of waiting to conclude the download of each 10s clips, we could instead feed data while downloading it. The fog has a promising role in this type of application, moving processing closer to the data sources can reduce dramatically the operation costs, saving computational resources, lift some pressure over the WAN and consequently be more energy friendly.

## 5 Conclusion

At this moment, the cloud model dominates, either for the fact of being the fastest growing industry (Costello, 2019) or for taking about 94% of workloads and compute instances by the year 2021 (Cisco, 2016). This model is here to stay as Bonomi states, and for this reason, is that fog computing appears as an extension of and not a replacement of the cloud model.

The rising of machinery that is becoming equipped with almost free computational resources, enable them to be connected to the WAN, over a wide variety range of things that affect directly our daily lives, such connected mattress, toasters, fridges, cooking machines, and others affecting indirectly. Mikko Hypponen, CRO at F-Secure, states that will be a day that will be cheaper buying devices with connectivity capabilities than the traditional ones, arguing that the value of data pushes to industry to this practice of equipping devices with connectivity capabilities, plus, those devices will not even need our Internet connection to transfer data back to the vendor cloud, using the 5G technology for that (Hypponen, 2017)

In overall perspective, 22 years ago, when IBM supercomputer Deep Blue won the chess game over the Kasparov, the supercomputer had a computing power of 11GFLOPS, the smartphone that we have used to stream the video has the computing power of 40.8GFLOPS, in other words, a 2019, 150€ phone could beat 3 “Kasparovs” at same time, and on top of that, it runs on battery. The development of computing power of those devices also increases, with that, fog computing

embraces the heterogeneity of hardware, meshing the network with different and all kind of nodes to offer ubiquitous computing to those nodes who need it.

Fog computing projects that the nodes aggregation can be horizontal or vertical and the combination of both has the potential of creating silos of resources for each vertical that is serving, and smart getaways (Aazam & Huh, 2014) can connect those silos, especially in information exchange, to create powerful insights. This vision is very appealing, but multiple bridges have to be built.

The academia is still researching the “where” and “how” questions, the “what” applications suitable are the easiest part to answer. The “where” questions, in our view, is the most difficult part, since involves physically devices and their ownership, consequently, the maintenance and security of those nodes by the owner part. In counterpart, if the big data centers have more demand than capacity, as the natural market reaction would be to increase the cloud prices, this increase on price can fuel the placement of these edge data centers. Another incentive could be the awareness of different business models that can originate from the owner of those edge data centers and lead entrepreneurs to invest in this computing model infrastructure. We have seen that the academia points a variety of sites like a local business, train stations, public transportation, telecom antennas, active network infrastructure and so on. The one that has a formal project and active development is the MEC from ETSI, with 28 publications covering a wide variety of topics from technical requirements to defining MAC APIs and with an ongoing proof of concept project, already available in Hackathons.

The “how” is the subsequent questions to be made, this new intermediary layer offers a new pool of computational resources, and as the word resource means, it must be used cost-effective. The academia studied several novel schemes to use those resources, moreover, how to merge those with the existing cloud infrastructure. Some of the reviewed works also propose ways to create stable resource pools, which they are mentioned as incentive-based schemes.

The SDN is mentioned in the real world as one real architectural candidate, we are stating this because we have South-Korea example. The implementation of 5G network for economic reasons led to the telecom operators to share underline physically infrastructure by using the SDN architecture. We saw it as a form of fog computing for the telecommunication vertical, where we have horizontality since they are multiple antennas and verticality, when as they own the antennas.

The SDN offers security by architectural design, however other papers reviewed in this work, security was not approached, and we have a recent example like Mirai malware, where the targets were only IoT devices, affecting 120K devices by using the common default passwords. The Mirai in 2016 targeted the second-largest DNS provider with a DDoS attack, making inaccessible websites like GitHub, Twitter, and other high-profile websites (Williams, 2016). This malware illustrates the “what” applications that can benefit from increasing of the connected devices, although not with the best intentions, and also show the network and data potential since this botnet has generated 1TB/s attack.

This unintended use of the connected devices does not affect the primary functionality of the respective device, but this offloaded task used some of the available computing power and some portion of the bandwidth to conduct the attack, meanwhile probing other potential vulnerable devices to be added to the botnet.

We have proposed an alternative to democratize device filiation, administrated in a single point. The platform enables an unbounded number of devices that can be part of the network, and users can deploy logic units on those nodes. We do not propose any business model, focusing on the technical part of the building the essential blocks to have a functional infrastructure. Nevertheless, we envision the creation of a market, where developers build pre-sets lambdas with some functionality and client buy those units, for example, someone creates a lambda (a lambda in our platform is a set of lambda(s)) for image classification, while other developer creates a lambda that efficiently stores images, and the client have some use case that receives images and wants to classify and store them, so, the client can buy those units and integrate into the pipeline at lambda creation. In an extreme case, with a little platform refractor, the client could only move the logic units without knowing anything about programing, representing only the business logic. The market perspective is part of the future work that will be studied.

In terms of the technical aspects and correlating with the fog computing properties, the platform can integrate geographically distributed devices with addressable lambda hosting bypassing technical connection difficulties for non-internet exposed devices via one bus where messages and commands can be exchanged. For those nodes that are Internet exposed, the communication is or can be done without the bus, and consumers can use this extra-calling mechanism when available. For simplicity and time constraints, we only support HTTP protocol,



however, how the platform is currently built, it lets internet-exposed devices with no isolation, to use any available port and within the lambda, the client can support any other protocol that we need. The client can follow real-time metrics like CPU and Memory, requests per second, and easily scaling horizontally each logic unit. The platform runtime is smart enough to only build one sandbox per lambda-node pair and not for instance-node pair, this way if the client increases their instances over the same lambda-node pair, the runtime only spawns another process using the same sandbox, making elastic computing fast and little overhead.

The technologies used to develop the essential platform blocks offer a great degree of scalability, both database, and the bus offer were built for distributed deployments in mind. The behavior of data distribution, even that was thoughtfully designed, deserves a study if the data is evenly distributed, plus, if the current queries that are being done perform well meanwhile the platform grows. In the bus, the results from the TODO application have shown that latency that this technology brings, either we do research another alternative or implement one, either way, this is a study that needs to be done in conjunction if latencies greater than 100ms from our potential users has a substantial impact on their operations.

The application tested served to make the point of having task offloading to the fog and to the cloud, the TODO app also has the purpose to benchmark the throughput of the current developed platform. The video application has the purpose to show a detailed view of the difficulties face for video analytics deployment making a comparison of the offloading video detection for the fog and cloud provider.

Summing up, the idea of the need of offloading tasks closer to the users, to offers immersive and new experiences like gaming on-demand and VR is emerging (Bilal & Erbad, 2017), other verticals also can take advantages to make smart and intelligent actions at or closer of edge nodes. Data-intensive (consumers side) and video streaming are the two main applications type that cloud models do not fit due to WAN constraints leaving the space for the emerging of this new computing model.

### **5.1.1 Other Related Works**

This dissertation work has resulted in open source contributions on two different projects: i) Rust Cassandra Driver and ii) Google's Firebase Java Admin. The first was already mentioned, we have implemented pagination for stateless and non-state applications. In this contribution

we have contributed with the implementation and with unit testing for the pagination feature, but also with code style uniformization across the project using the “Rust fmt” tool.

The second contribution was in Google’s Firebase Java admin (Gonçalves, 2019). This library offered by Google is to administrate programmatically Firebase services using Java programming language, such as Android mobile notifications, real-time databases, ML Kit and so on. We introduced one API simplification over the Messaging product (related to Android & iOS mobile notifications). The scenario of sending messages in batch mode was already provided by the library, in other words, the concept of sending the same message for multiples targets (devices) was already implemented in the class “MulticastMessage”, however the developer (the one who use the library) to use this feature will pass an object of the “List” data type. The “List” data type in Java allows repeated elements, so when the developer passes a list he is saying, in the code alone, that may have targets that can receive the same notification because the data type allows it. Our improvement was to change the argument data type from the “List” data type to “Collection” data type, with this, the developer can have the targets in a “Set” data type, and the data type, by itself, express clearly the developer intention. The proposal was accepted and already has two minor releases with that improvement. This contribution appeared when we are building one Android runtime for the solution presented in this work.

## Bibliography

Aazam, M. & Huh, E.-N., 2014. *Fog Computing and Smart Gateway Based Communication for Cloud of Things*. [Online]

Available at: <https://doi.org/10.1109/ficloud.2014.83>

[Accessed 12 10 2019].

Aazam, M., Zeadally, S. & Harras, K. A., 2018. Offloading in fog computing for IoT: Review, enabling technologies, and research opportunities. *Future Generation Computer Systems*, , 87(), pp. 278-289.

Alphabet, 2018. *Efficiency: How we do it – Data Centers – Google*. [Online]

Available at: <https://www.google.com/about/datacenters/efficiency/internal/>

[Accessed 11 1 2019].

Amazon, 2019. *AWS Snowball*. [Online]

Available at: <https://aws.amazon.com/snowball/>

[Accessed 2019].

Amazon, 2019. *AWS Snowmobile*. [Online]

Available at: <https://aws.amazon.com/snowmobile/>

[Accessed 2019].

Ananthanarayanan, G. et al., 2017. *Real-time Video Analytics – the killer app for edge computing*. [Online]

Available at: [https://www.microsoft.com/en-us/research/wp-](https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/CO_COMSI-2017-03-0045.R1_Ananthanarayanan.pdf)

[content/uploads/2017/06/CO\\_COMSI-2017-03-0045.R1\\_Ananthanarayanan.pdf](https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/CO_COMSI-2017-03-0045.R1_Ananthanarayanan.pdf)

- Anon., . *Developer Guide – Protocol Buffers – Google Code*. [Online]  
Available at: <https://code.google.com/apis/protocolbuffers/docs/overview.html>  
[Accessed 11 10 2019].
- Apache Cassandra, 2019. *Apache Cassandra*. [Online]  
Available at: <http://cassandra.apache.org>
- Apache Marathon, 2019. *Marathon*. [Online]  
Available at: <https://mesosphere.github.io/marathon/>
- Astuto, B. N. et al., 2014. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *Communications Surveys and Tutorials*, Volume 16, pp. 1617-1634.
- Bawa, M. et al., 2003. Peer-to-peer research at Stanford. *Sigmod Record*, , 32(3), pp. 23-28.
- Bilal, K. & Erbad, A., 2017. *Edge computing for interactive media and video streaming*. [Online]  
Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7946410>  
[Accessed 4 10 2019].
- Bonomi, F., Milito, R., Natarajan, P. & Zhu, J., 2014. Fog Computing: A Platform for Internet of Things and Analytics. *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp. 169-186.
- Bonomi, F., Milito, R., Zhu, J. & Addepalli, S., 2011. Fog computing and its role in the internet of things. p. 13–16.
- Bosh, 2016. *Which camera settings influence Video Analytics performance and why?*. [Online]  
Available at:  
[http://resource.boschsecurity.us/documents/TN\\_VCA\\_camera\\_settin\\_WhitePaper\\_enUS\\_24087849739.pdf](http://resource.boschsecurity.us/documents/TN_VCA_camera_settin_WhitePaper_enUS_24087849739.pdf)  
[Accessed 2019].
- Cable, 2019. *Worldwide mobile data pricing: The cost of 1GB of mobile data in 230 countries*. [Online]  
Available at: <https://www.cable.co.uk/mobiles/worldwide-data-pricing/>
- Cisco, 2016. *Cisco Global Cloud Index: Forecast and Methodology, 2016–2021*, San Jose: Cisco.
- Cisco, 2016. *Platform Support Matrix*. [Online]  
Available at: <https://developer.cisco.com/docs/iox/#!/platform-support-matrix/platform-support-matrix>  
[Accessed 24 1 2019].
- Costello, K., 2019. *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019*. [Online]

Available at: <https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g>

Craciunescu, R. et al., 2015. Implementation of fog computing for reliable e-health applications. *IEEE*, p. 459–463.

Culjak, I. et al., 2012. *A brief introduction to OpenCV*. [Online]  
Available at: <https://ieeexplore.ieee.org/document/6240859>  
[Accessed 11 10 2019].

Dewulf, K., 2013. *Sustainable Product Innovation: The Importance of the Front- End Stage in the Innovation Process*. [Online]  
Available at: <https://intechopen.com/books/advances-in-industrial-design-engineering/sustainable-product-innovation-the-importance-of-the-front-end-stage-in-the-innovation-process>  
[Accessed 13 10 2019].

Dunbar, B., 2018. *Disruption Tolerant Networking*. [Online]  
Available at: <https://www.nasa.gov/content/dtn>  
[Accessed 2019].

EnerData, 2018. *Global Energy Trends*, France: EnerData.

Enescu, M., 2014. *From Cloud to Fog Computing and IoT*. Chicago, LinuxCon + CloudOpen.

Evans, P. C. & Annunziata, M., 2012. *Industrial Internet: Pushing the Boundaries of Minds and Machines*, s.l.: Imagination at Work.

Evans, R. & Gao, J., 2016. *DeepMind AI Reduces Google Data Centre Cooling Bill by 40%*. [Online]  
Available at: <https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/>  
[Accessed 11 1 2019].

Farrell, K. F. a. S., 2008. DTN: an architectural retrospective. *IEEE Journal on Selected Areas in Communications*, Volume 26, pp. 828-836.

Flamm, K., 2018. Measuring Moore's Law: Evidence from Price, Cost, and Quality Indexes. *National Bureau of Economic Research*, , (), p. .

Fricker, C., Guillemin, F., Robert, P. & Thompson, G., 2016. Analysis of an Offloading Scheme for Data Centers in the Framework of Fog Computing. *ACM Trans. Model*, p. 16.

Giacaglia, G., 2019. [Online]  
Available at: <https://hackernoon.com/data-is-the-new-oil-1227197762b2>  
[Accessed 2019].

- Gilbert, S. & Lynch, N. A., 2012. *Perspectives on the CAP Theorem*. [Online]  
Available at: <https://dspace.mit.edu/openaccess-disseminate/1721.1/79112>  
[Accessed 11 9 2019].
- Gonçalves, N., 2018. *Linux Namespaces*. [Online]  
Available at: [https://github.com/NunuM/linux\\_namespaces\\_tutorial](https://github.com/NunuM/linux_namespaces_tutorial)  
[Accessed 2018].
- Gonçalves, N., 2019. *Code clarity on MulticastMessage class*. [Online]  
Available at: <https://github.com/firebase/firebase-admin-java/issues/280>
- Gonçalves, N., 2019. *Pager for stateless executions*. [Online]  
Available at: <https://github.com/AlexPikalov/cdrs/pull/258>  
[Accessed 2019].
- Google, 2019. *Protocol Buffers*. [Online]  
Available at: <https://developers.google.com/protocol-buffers>
- Guilfoyle, M., 2018. *Growing Role of Edge Intelligence in IoT*, Dedham, Massachusetts: ARC.
- Habak, K., Ammar, M. H., Harras, K. A. & Zegura, E. W., 2015. *Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge*. [Online]  
Available at: <http://cc.gatech.edu/~khabak3/papers/femtocloud-cloud'15.pdf>  
[Accessed 17 2 2019].
- Hasan, R., Hossain, M. M. & Khan, R., 2017. Aura: An incentive-driven ad-hoc IoT cloud framework for proximal mobile computation offloading. *Future Generation Computer Systems*, , 86(), pp. 821-835.
- HashiCorp, 2019. *Vagrant*. [Online]  
Available at: <https://www.vagrantup.com>
- Hashimoto, M., . *Vagrant 1.6*. [Online]  
Available at: <https://www.hashicorp.com/blog/vagrant-1-6.html>  
[Accessed 12 10 2019].
- Hegde, Z., 2018. *Predictive Maintenance – What you need to know*. [Online]  
Available at: <https://www.iot-now.com/2018/05/02/81526-predictive-maintenance-need-know/>  
[Accessed 2019].
- Hindman, B. et al., 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *USENIX*.
- Hobbs, T., 2019. *CQL BINARY PROTOCOL v4*. [Online]  
Available at: [https://github.com/apache/cassandra/blob/trunk/doc/native\\_protocol\\_v4.spec](https://github.com/apache/cassandra/blob/trunk/doc/native_protocol_v4.spec)  
[Accessed 2019].

- Hughes, I., 2017. *FogHorn gets funding boost and injects machine learning deep in IIoT devices*, s.l.: 451 Research.
- Huh, M. A. a. E., 2015. Dynamic resource provisioning through Fog micro datacenter. *IEEE*, pp. 115-110.
- Hypponen, M., 2017. *Mikko Hypponen (F-Secure) on Internet of Insecure Things | TNW Conference 2017*. [Online]  
Available at: <https://www.youtube.com/watch?v=aP6aUQjERBs&t=63s>
- IBM, 2019. *Camera frame rate, resolution, and video format requirements*. [Online]  
Available at:  
[https://www.ibm.com/support/knowledgecenter/en/SS88XH\\_1.6.1/iva/install\\_planning\\_video\\_source\\_requirements.html](https://www.ibm.com/support/knowledgecenter/en/SS88XH_1.6.1/iva/install_planning_video_source_requirements.html)
- IHS, 2016. *Internet of things (iot) connected devices installed base world- wide from 2015 to 2025*. s.l.:s.n.
- Iorga, M. et al., 2018. Fog Computing Conceptual Model: Recommendations of the National Institute of Standards and Technology. *National Institute of Standards and Technology*, Volume 500.
- King, D., 2018. *FogHorn*. [Online]  
Available at: <https://blog.foghorn.io/david-king-on-the-future-of-iiot-edge-intelligence>  
[Accessed 2019].
- Klein, D., 2008. *History of Digital Storage*, s.l.: Micron Technology, Inc..
- Kocić, J., Jovičić, . & Drndarević, V., 2019. An End-to-End Deep Neural Network for Autonomous Driving Designed for Embedded Automotive Platforms. *Sensors (Bazel)*.
- Koen, P. et al., 2001. Providing Clarity and A Common Language to the “Fuzzy Front End”. *Research-Technology Management*, 44(2), pp. 46-55.
- Kurkowski, S. H., Camp, T. & Colagrosso, M., 2005. MANET simulation studies: the incredibles. *Mobile Computing and Communications Review*, , 9(4), pp. 50-61.
- L. Velmurugan, G. R. a. M. J. S., 2017. Google Glass Based GPS Navigation Using Smartwatch. *IEEE International Conference on Computational Intelligence and Computing Research*, pp. 1-5.
- Lampert, L., 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2), pp. 133-169.
- Lema, M. et al., 2017. Business Case and Technology Analysis for 5G Low Latency Applications. *IEEE Access*.
- Liang, K., Zhao, L., Chu, X. & Chen, H.-H., 2017. An Integrated Architecture for Software Defined and Virtualized Radio Access Networks with Fog Computing. *IEEE Network*, pp. 80-87.

- Littledata, 2019. *What is the average time before full desktop page load?*. [Online]  
Available at: <https://www.littledata.io/average/time-before-full-desktop-page-load>  
[Accessed 2019].
- Liu, Y. et al., 2017. Incentive mechanism for computation offloading using edge computing: A Stackelberg game approach. *IEEE*, p. 399–409.
- Luan, T. H. et al., 2015. Fog Computing: Focusing on Mobile Users at the Edge. *ArXiv preprint*.
- Meurisch, C. et al., 2017. Decision Support for Computational Offloading by Probing Unknown Services. *ICCCN*, pp. 1-9.
- Moore, G. E., 1965. Cramming more components onto integrated circuits. *Electronics*, Volume 38.
- Nebbiolo Technologies, 2018. *Fog vs Edge Computing*. [Online]  
Available at: <https://www.nebbiolo.tech/wp-content/uploads/whitepaper-fog-vs-edge.pdf>  
[Accessed 05 2019].
- Nielsen, H. & LaLiberte, D., 1999. *Detecting the Lost Update Problem Using Unreserved Checkout*. [Online]  
Available at: <https://www.w3.org/1999/04/Editing/>
- Nielsen, J., 2018. *Nielsen's Law of Internet Bandwidth*. [Online]  
Available at: <https://www.nngroup.com/articles/law-of-bandwidth/>  
[Accessed 22 1 2019].
- OpenCV, 2019. *OpenCV*. [Online]  
Available at: <https://opencv.org>
- OpenFog, 2017. *OpenFog Reference Architecture for Fog Computing*, s.l.: OpenFog Consortium.
- Patel, M. et al., 2014. Mobile Edge Computing Introductory Technical White Paper. p. 36.
- Pu, L., Chen, X., Xu, J. & Fu, X., 2016. D2D Fogging: An Energy-Efficient and Incentive-Aware Task Offloading Framework via Network-assisted D2D Collaboration. *IEEE Journal on Selected Areas in Communications*, , 34(12), pp. 3887-3901.
- RabbitMQ, 2019. *Messaging that just works — RabbitMQ*. [Online]  
Available at: <https://www.rabbitmq.com>
- Roach, J., 2018. *Under the sea, Microsoft tests a datacenter that's quick to deploy, could provide internet connectivity for years*. [Online]  
Available at: <https://news.microsoft.com/features/under-the-sea-microsoft-tests-a-datacenter-thats-quick-to-deploy-could-provide-internet-connectivity-for-years/>  
[Accessed 11 1 2019].



Satyanarayanan, M., Bahl, P., Caceres, R. & Davies, N., 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, Volume 8, pp. 14-23.

Satyanarayanan, M., Bahl, P., Caceres, R. & Davies, N., 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE*, pp. 4-8.

Srivastava, P. & Khan, R., 2018. A Review Paper on Cloud Computing. *International Journal of Advanced Research in Computer Science and Software Engineering*, Volume 8.

Tesla, 2019. *Annual Shareholder Meeting*. [Online]  
Available at: [https://www.tesla.com/pt\\_PT/shareholdermeeting](https://www.tesla.com/pt_PT/shareholdermeeting)  
[Accessed 2019].

Valancius, V. et al., 2009. Greening the Internet with Nano Data Centers. *ACM*, pp. 37-48.

Warthman, F., 2015. *Delay- and Disruption-Tolerant Networks*. [Online]  
Available at: [http://ipnsig.org/wp-content/uploads/2015/09/DTN\\_Tutorial\\_v3.2.pdf](http://ipnsig.org/wp-content/uploads/2015/09/DTN_Tutorial_v3.2.pdf)  
[Accessed 2019].

Williams, C., 2016. *Today the web was broken by countless hacked devices – your 60-second summary*. [Online]  
Available at: [https://www.theregister.co.uk/2016/10/21/dyn\\_dns\\_ddos\\_explained/](https://www.theregister.co.uk/2016/10/21/dyn_dns_ddos_explained/)

Winans, M., 2017. *10 Key Marketing Trends for 2017*, s.l.: IBM Marketing Cloud.

Zhang, K. et al., 2016. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE*, p. 5896–5907.

Zhao, X., Zhao, L. & Liang, K., 2016. *An Energy Consumption Oriented Offloading Algorithm for Fog Computing*. [Online]  
Available at: [https://link.springer.com/chapter/10.1007/978-3-319-60717-7\\_29](https://link.springer.com/chapter/10.1007/978-3-319-60717-7_29)  
[Accessed 17 2 2019].

