2018

On the Effectiveness of an IOT - FOG - CLOUD Architecture for a real-world application

Nathan Wheeler
University of North Florida, n00871989@ospreys.unf.edu

ON THE EFFECTIVENESS OF AN IOT - FOG - CLOUD ARCHITECTURE FOR A REAL-WORLD APPLICATION

by

Nathan Wheeler

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computing and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

August, 2018

The thesis "On the Effectives of an IoT – Fog – Cloud Architecture for a Real-World Application" submitted by Nathan Wheeler in partial fulfillment of the requirements for the degree of Master of Science in Computing and Information Sciences has been

Approved by the thesis committee: Date

_____ _____

Dr. Sanjay P. Ahuja
Thesis Advisor and Committee Chairperson

_____ _____

Dr. Roger Eggen

_____ _____

Dr. Karthikeyan Umapathy

Accepted for the School of Computing:

_____ _____

Dr. Sherif Elfayoumy
Director of the School

Accepted for the College of Computing, Engineering and Construction:

_____ _____

Dr. William Klostermeyer
Dean of the College

Accepted for the University:

_____ _____

Dr. John Kantner
Dean of the Graduate School

ACKNOWLEDGEMENT

I'd like to thank my friends Aaron Jones and Phillip Kosmowski for their unwavering friendship through my years of studies culminating in this thesis as well as Paul Kosmowski for his brilliance in helping to discover an excellent use-case for an IoT-Fog-Cloud architecture. I wish to thank my mother for providing me with my guiding life principles, helping me to get this far, as well as my girlfriend Kendall Durante for her love and support throughout this research process.

I thank my thesis advisor Dr. Sanjay P. Ahuja for his expertise advice, feedback, and suggestions. I also thank Dr. Roger Eggen and Dr. Karthik Umapathy who agreed to be on my thesis committee and provided great feedback in this research process. Working with these three distinguished UNF faculty members has been a great honor and a privilege. I'll also thank the great minds who've come before us, who paved the way for the scientific community to be farther along the journey than we could've otherwise been.

CONTENTS

List of Figures ix

List of Tables xi

Abstract xii

Chapter 1: Introduction 1

1.1 Research Objectives 2

Chapter 2: Literature Review 4

Chapter 3: Research Methodology 9

3.1 Application Description 9

3.2 Application 1: Fog-Enabled Smart Coaster 10

3.2.1 IoT-Level Activities 10

3.2.2 Fog-Level Activities 10

3.2.3 Cloud-Level Activities 11

3.3 Application 2: Fogless Smart Coaster 11

3.4 Technology Stack 11

3.4.1 IoT-Level 12

3.4.2 Fog-Level 13

3.4.3 Cloud-Level 16

3.5 Application Technical Overview 18

3.6 AWS Greengrass 19

3.6.1 Lambda Functions on AWS Greengrass 20

3.6.2 Device Creation in AWS IoT 21

3.6.3 Subscription Configuration 22

3.6.4 Configuration and Lambda Deployment 23

3.6.5 Discovering the Greengrass Core 23

3.6.6 Sending Data from an IoT Device to the Greengrass Core 24

3.7 Application Demonstration 24

3.8 Amazon Machine Learning 26

3.8.1 Creating a New Datasource 27

3.8.2 Creating a New Machine Learning Model 28

3.8.3 Evaluating the Machine Learning Model 28

3.8.4 Generating Predictions with the Machine Learning Model 31

Chapter 4: Metrics And Specifications 33

4.1 Metrics 33

4.1.1 One-way Transfer Time (Milliseconds) 33

4.1.2 Round-Trip Transfer Time (Seconds) 34

4.2 Specifications 34

4.2.1 Hardware Specification 35

4.2.2 Network Specification 36

4.2.3 Software Specification 37

Chapter 5: Test Methodology 38

5.1 Application Requests 38

5.2 Transfer Time Study Measurements 38

Chapter 6: Results and Analysis 40

6.1 Study on Application Response Times 40

6.2 Lessons Learned 44

6.2.1 AWS Greengrass 45

6.2.2 Amazon Machine Learning 46

Chapter 7: Conclusions and Future Work 47

7.1 Conclusions 47

7.2 Future work 48

References 49

Appendix A: AWS Greengrass Installation, Configuration and Execution 52

Environment Setup for Greengrass 52

Installing the Greengrass Core Software 52

Start AWS Greengrass on the Core Device 53

Lambda Functions on AWS Greengrass 56

Device Creation in AWS IoT 58

Subscription Configuration 59

Lambda Deployment 60

Discovering the Greengrass Core 60

Sending IoT Data to the Greengrass Core 62

VITA 65

FIGURES

Figure 1: Sequence Diagram for Fogless Application 18

Figure 2: Sequence Diagram for Fog-Enabled Application 19

Figure 3: AWS Greengrass Interaction Model 20

Figure 4: Lambda Functions on Greengrass Step 4 21

Figure 5: Thesis Greengrass Subscriptions 22

Figure 6: Fog Single Coffee Shop GUI 25

Figure 7: Cloud Single Coffee Shop GUI 25

Figure 8: Cloud Manager View GUI 26

Figure 9: Updated ML Model Performance 29

Figure 10: Amazon ML "Confusion Matrix" 29

Figure 11: Description of Amazon's F1 Value 30

Figure 12: Generating Batch Predictions Part 5 31

Figure 13: Batch Prediction Results 32

Figure 14: Box Plot for 1-Way Data Transfer Statistics 42

Figure 15: Box Plot for Round-Trip Transfer Time Statistics 44

Figure 16: Greengrass Start Output 55

Figure 17: Greengrass Core Daemon Check Output 55

Figure 18: Greengrass config.json File 56

Figure 19: Lambda Functions on Greengrass Step 4 57

Figure 20: Example Pub/Sub Script Part 1 62

Figure 21: Example Pub/Sub Script Part 2 63

Figure 22: Example Pub/Sub Script Part 3 64

TABLES

Table 1: One-Way Transfer Time in Milliseconds 40

Table 2: One-Way Transfer Time Statistics 41

Table 3: Round-Trip Transfer Time in Seconds 41

Table 4: Round-Trip Transfer Time Statistics 42

ABSTRACT

Fog Computing is an emerging computing paradigm that shifts certain processing closer to the Edge of a network, generally within one network hop, where latency is minimized, and results can be obtained the quickest. However, not a lot of research has been done on the effectiveness of Fog in real-world applications. The aim of this research is to show the effectiveness of the Fog Computing paradigm as the middle layer in a 3-tier architecture between the Internet of Things (IoT) and the Cloud. Two applications were developed: one utilizing Fog in a 3-tier architecture and another application using IoT and Cloud with no Fog. A quantitative and qualitative analysis followed the application development, with studies focused on application response time and walkthroughs for AWS Greengrass and Amazon Machine Learning.

Furthermore, the application itself demonstrates an architecture which is of both business and research value, providing a real-life coffee shop use-case and utilizing a newly available Fog offering from Amazon known as Greengrass. At the Cloud level, the newly available Amazon Machine Learning API was used to perform predictive analytics on the data provided by the IoT devices. Results suggest that Fog-enabled applications have a much lower range of response times as well as lower response times overall. These results suggest Fog-enabled solutions are suitable for applications which require network stability and reliably lower latency.

Chapter 1

INTRODUCTION

The Internet of Things (IoT) is an emerging paradigm in which objects, which would otherwise be unable to gather and transmit data, are empowered with sensors and/or actuators along with computing and networking resources, enabling these devices to communicate with other devices directly or over the Internet. Leading industry researchers at Gartner estimate that 20 billion devices will be connected to the Internet of Things by 2020 [Gartner17]. According to global intelligence firm IDC, spending on IoT will increase from $936 billion to $1.4 trillion in the next three years [IDC17]. Cloud Computing is another emerging paradigm, which provides users with a virtually endless supply of computing resources in addition to powerful APIs for Machine Learning and many more tools which would be otherwise prohibitively expensive to implement in a traditional data center.

IoT and Cloud Computing was a natural pairing, enabling advanced predictive analytics, historical data analysis and automation, the likes of which the world has never seen before. Despite the tremendous potential of this integration, some interesting questions have arisen which require careful consideration. For example, how does one minimize time between when events occur and when one can act on the data produced by those events? Many times, the best place to act on data is at the “Edge” of the network where

the events occur. To solve these problems and many more, another new computing paradigm has been introduced under the name “Fog Computing” [Cisco15]. Fog Computing provides compute resources at the edge of the network to act on local events in real-time. The main goal of this research is to demonstrate Fog’s effectiveness as a middle tier in a 3-tier architecture with IoT and Cloud Computing.

1.1 Research Objectives

The expected contributions of this thesis are as follows:

1. Demonstrate the effectiveness of Fog as a platform for IoT and Cloud applications.
2. Develop an application, whose architecture and implementation are of both research and business value.
3. Provide a comparison and analysis of an example application which implements a Fog-enabled IoT and Cloud architecture with that of an application which implements a Fogless IoT and Cloud architecture.
4. Provide researchers with a better understanding of the insufficiently-researched Amazon Machine Learning Cloud API by utilizing it for predictive analytics and documenting the experience.
5. Provide researchers a better understanding of the new AWS Greengrass software by utilizing it at the Fog-level and documenting the experience.

Despite Fog’s emergence to improve performance in distributed applications, little research has been done on the practical performance improvements. Further,

implementations of Fog in the Literature have been limited to a few use-cases, generally relating to smart homes, cars or cities. With this thesis, the aim is to provide researchers and industry practitioners insight into how latency can realistically be improved with Fog and provide a unique and practical implementation of an IoT-Fog-Cloud architecture which are sparse in the literature now.

Chapter 2

LITERATURE REVIEW

Fog is defined as an architecture that distributes computation, communication, control and storage closer to the end users along the cloud-to-things continuum [Chiang16]. Chiang and Zhang mention that "Fog" is sometimes used interchangeably with the term "Edge", although Fog is broader than Edge. The authors list many benefits of Fog and how it complements both IoT and Cloud computing. The authors also cite Fog use case studies such as content caching at the edge and edge analytics.

In the future works section, the authors stated that the Fog will create new opportunities for us to design end-to-end systems to achieve better tradeoffs between distributed and centralized architectures. The authors also stated, to propel Fog research forward, there is a need for academia to interact more closely with industry. It is for this reason of helping to bridge the gap between business and academia, and to propel Fog research forward that the approach of a real-world-application-based research approach was taken in this thesis.

Chieochan et al. prototyped a smart mushroom farm at Maejo University[Chieochan17]. Humidity data was collected, then the sprinkler and Fog pumps were initiated when the humidity hit a certain threshold. Humidity data was also sent to the Cloud where it was visualized on computers and mobile devices. The watering status could also be viewed on

the Cloud. This research, completed fairly early in the development of Fog Computing, demonstrated an excellent real-world use-case of an IoT-Fog-Cloud architecture. The author's research methodology consisted of a 5-step plan-driven development cycle: Requirement and Feasibility Study, System Analysis and Design, Implementation, System Validation and Maintenance. Inspiration was taken from this method of development for Fog applications as well as their method of taking actions on events once a sensor passed a certain threshold.

Dutta and Roy presented an IoT-Fog-Cloud based architecture for a smart building [Dutta17]. In the IoT Layer, sensing and communication capabilities are incorporated into the physical things of the building such as the doors, lights, fans, air conditioner, garage,water and pumping systems, air purifier, fire alarm and elevator . In the Fog layer, the Fog server worked as both an internet gateway and made some decisions based on sensor data. In the Cloud layer, data from the Fog layer was forwarded to a Cloud database which is maintained by a Cloud server, which was then used for controlling the devices from a Smart Phone or PC.

Dutta and Roy's research provided a real-world example of a working, three-tier, IoT, Fog and Cloud architecture which provided a great reference when designing the application architecture for this thesis research [Dutta17]. Furthermore, these authors were able to identify certain IoT events in their application which could be handled entirely on the Fog, which enabled a faster scenario than waiting for a response from the Cloud.

Giang et al. examined the development of IoT applications from the Fog Computing paradigm [Giang15]. The authors mention that, in the Fog paradigm, computing infrastructure at the network edge in devices and gateways is leveraged for efficiency and timeliness. The authors go on to propose a Distributed Dataflow (DDF) programming model for IoT that utilizes computing infrastructure aross the Fog and the Cloud. The programming language aims to address the problems of heterogeneous devices and resources, a tightly coupled perception-action cycle and widely distributed devices and processing. The authors conclude by suggesting that their DDF programming model provides an easy way to design and devleop IoT applications by combining application constraints and device capabilities to help drive the dynamic deployment of application logic. The insight provided by these authors shed light on certain environmental challenges which enable Fog to be effective. These challenges include scalability, heterogeneity of devices, timeliness requirements and mobility.

Husni et al. built a smart car using IoT technology [Husni16]. IoT data was sent to IBM BlueMix to obtain predictive maitennace information as well as to provide monitoring for the car. The IoT information was transmitted from the car to a smartphone via bluetooth and from the phone to the Cloud using a celluar internet connection. The use of the three-tiered architecture utilizing machine learning in the Cloud by Husni et al. was the inspiration for utilizing Amazon Machine Learning for predictive analytics in this paper [Husni16]. Authors seem to recognize that there are certain workloads, such as machine learning, that are more suited toward the Cloud, making the Cloud a key component in the three-tiered IoT, Fog and Cloud architecture.

Zhuo Chen et al. used the idea of a “cloudlet”, which seems to be identical in concept to a fog but is described as a “datacenter in a box” [Zhuo Chen16]. This cloudlet was used for parallelized computation offloading from a prototype Google Glass system. The entire system is known as “Gabriel” and aims to assist cognitively impaired individuals to recognize other people by using Google Glass’s image capture feature along with image recognition technology. Exploiting cloudlet parallelism helped to improve the performance of Gabriel, but the computational workloads pushed the Google Glass’s battery life down to an unsustainable two hours.

The Gabriel project provides yet another great example of a real-world application which utilizes IoT, Fog and Cloud together in an effective way. This research provided an example of one of the limitations that IoT Devices can have which is a high degree of energy consumption.

All the authors mentioned thus far have added a necessary source of inspiration for the creation of this thesis. However, after a thorough review of both the Association for Computing Machinery (ACM) and IEEE databases, very few papers have provided metrics which show the kinds of performance increases (lowered response times) to be expected in real-world applications which are enabled with Fog. This thesis was built to help fill the gap which was observed in the literature. As such, in building multiple applications, one with Fog and one without Fog, we were able to analyze the difference in application response times under a real-world load and determine that the Fog-enabled architecture was more performant. As an additional research component, this thesis aims

to familiarize researchers with AWS Greengrass and Amazon Machine Learning. This decision was made after choosing to utilize these two technologies in the design of the application architecture at the Fog and Cloud level respectively and finding no mention of these technologies in the literature.

Chapter 3

RESEARCH METHODOLOGY

3.1 Application Description

The use-case chosen for this research is that of a coffee shop. Temperature and volume sensors placed in coffee cups allow temperature and volume data for all customers in the coffee shop to be viewed by the servers via a centrally located Graphical User Interface (GUI). The data is sent to the Cloud where managers of a group of coffee shops can view all their coffee shops simultaneously and can get predictions from Amazon Machine Learning about which coffee shop will perform better in terms of refilling customer beverages. Two versions of the application were developed: A “Fog-enabled” version, in which a Fog node serves as the middle tier in a 3-tier IoT-Fog-Cloud architecture, and a “Fogless” version, in which only IoT and Cloud are utilized in a 2-tier architecture. Studies on the response times of these two applications under a real-life workload provide insights into the benefits of an Iot-Fog-Cloud architecture and server as the basis for the thesis results.

3.2 Application 1: Fog-Enabled Smart Coaster

The Fog-enabled application utilizes fog to enhance performance. This application is split into three levels: IoT, Fog and Cloud. Activities performed at each level are described in the following three sections.

3.2.1 IoT-Level Activities

Temperature sensors detect when a customer's coffee is getting too cold. Weight sensors calculate volume data to calculate when a customer's drink is running low. This data is sent to the local Fog server for consideration.

3.2.2 Fog-Level Activities

Temperature and volume data are displayed in a local monitor within the coffee shop, which shows a view of all drinks in the coffee shop, their current volume, a regular coffee cup or a blue or blue coffee cup depending on if the drink is too cold or just right and a bubble message above the cup if the drink is running low, to remind the server that a refill is needed.

3.2.3 Cloud-Level Activities

Coffee shop data are sent to the Cloud, where higher-level managers can see all drinks in all their coffee shops in real time. In addition, historical data are stored and analyzed with machine learning algorithms to predict which of the manager's coffee shops' staff will perform better in the future, to inform and improve business operations.

3.3 Application 2: Fogless Smart Coaster

The Fogless smart coaster application performs the same functions as the Fog-enabled application, but all processing and storage previously done on the Fog is now necessarily done on the Cloud. This necessitates a remote connection into the Cloud server from the coffee shop to view coffee shop-specific IoT data.

3.4 Technology Stack

The technology stack represents the programs and technologies which were used throughout the different layers of the application's architecture. The technology stack utilized for the Fog-enabled application is described by level. The three levels are IoT, Fog and Cloud. The components for these three levels can be found in sections 3.4.1, 3.4.2 and 3.4.3 respectively.

3.4.1 IoT-Level

The IoT-level of the application architecture contains both hardware and software packages either attached to or installed on the IoT device as well as the IoT device itself. The components utilized at the IoT-level are described in the following subsections.

3.4.1.1 Raspberry Pi

The Raspberry Pi is being used in conjunction with the DS18b20 temperature sensor to measure the temperature data from the coffee cups in the shop. In addition, the Pi is generating simulated volume data as well simulating semi-random customer drinking rates and server refilling rates to facilitate the gathering of data and ingestion into Amazon Machine Learning.

3.4.1.2 Raspbian Jessie

Raspbian is a Debian-based operating system optimized for the Raspberry Pi device [Raspberry Pi Foundation18]. Jessie is the name of the latest distribution of Raspbian.

3.5.1.3 AWS Python IoT Device SDK with built-in Greengrass discovery.

This SDK allows the Raspberry Pi to Discover the Fog device, publish and subscribe to topics using the Message Queueing Telemetry Transport (MQTT) protocol and use other built-in Amazon Web Services (AWS) functionality to interact with the Cloud.

3.4.1.4 DS18b20 Digital Temperature Sensor

The DS18b20 is used in conjunction with the Raspberry Pi, a 3.7-ohm resistor, a breadboard jumper leads from the sensor to the Raspberry Pi to measure the temperature of the customer coffee cups in the coffee shop scenario. The guide which was followed for the setup of the sensor can be found on the YouTube channel, “Pi My Life Up” [Pi My Life Up16].

3.4.2 Fog-Level

The Fog-level of the application architecture contains software packages installed on the Fog Server as well as the Fog server itself. The components utilized at the Fog-level are described in the following subsections.

3.4.2.1 AWS Greengrass Core

Greengrass Core is being used as a Fog software package, installed on a local Ubuntu 16.04 LTS virtual machine. Greengrass Core allows IoT devices to interact locally with the Greengrass core device, enabling lightning-fast response times, security of data and easy integration with AWS. AWS Greengrass are discussed in greater detail in its own section, AWS Greengrass.

3.4.2.2 AWS Lambda

AWS Lambda functions are the code which are deployed onto Amazon Greengrass; these functions can either be long-standing or triggered based on some event such as a device publishing data to a topic via a message broker, to which other devices are subscribed in a publisher-subscriber (pub-sub) interaction model.

3.4.2.3 Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world [Node.js Foundation18A].

3.4.2.4 Express.js

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications [Node.js Foundation18B]. Express.js is the framework on which the GUI server and RESTful API is built.

3.4.2.5 jQuery/HTML/CSS

These are the tools which are used to build the GUI. HTML and CSS are used for building and styling the GUI, and jQuery handles all data dynamic functionality such as updating the Document Object Model (DOM) when a request is received from the Raspberry Pi to show the temperature of the coffee cup on the screen.

3.4.2.6 Ubuntu 16.04 LTS

Ubuntu is an open source software operating system that runs from the desktop, to the cloud, to all your internet connected things [Canonical Ltd.18]. Ubuntu is the operating system on which the project's Fog system is built.

3.4.2.7 Oracle VM Virtualbox

Virtualbox is a hypervisor which allows us to create the virtual machine needed to serve as a Fog node in the Fog-Enabled architecture.

3.4.3 Cloud-Level

The Cloud-level of the application architecture contains software packages installed on the Cloud Server, the Cloud server itself as well as external services offered by Amazon Web Services. The components utilized at the Cloud-level are described in the following subsections.

3.4.3.1 Amazon EC2

Amazon EC2 provides virtualized servers as a service in the Cloud. One server has been allocated to both serve as the RESTful endpoint for the Fog data and to host the Manager View of the GUI, which tracks all coffee shops under a manager's jurisdiction.

3.4.3.2 Amazon S3

Amazon S3 provides object storage in the Cloud. S3 is necessary because only data from S3 can be ingested into Amazon Machine Learning for batch predictions.

3.4.3.3 Ubuntu 16.04 LTS

To maintain an apples-to-apples comparison, we're using the same operating system on both the Fog server and the Cloud server.

3.4.3.4 Node.js and Express.js

Both Node.js and Express.js were used to build the RESTful endpoint for the Fog node, the Manager GUI and the single coffee shop view for the Fogless application.

3.5.3.5 jQuery/HTML/CSS

These web programming tools were used again at the Cloud level to build the GUI for the Manager View on top of the Node.js and Express.js frameworks.

3.4.3.6 Amazon Machine Learning

Amazon Machine Learning is AWS's Cloud API which aims to provide developer-friendly Machine Learning algorithms at a low cost without a steep learning curve. In this project, Amazon Machine Learning is used to predict which coffee shop will perform better based on past server performance data gleamed from IoT device data.

3.5 Application Technical Overview

Figure 1 illustrates the high-level activities that occur in the Fogless Application. IoT Sensor data is collected from the sensors attached to the Raspberry Pi and is sent to the Cloud Server. The Cloud Server then updates the Cloud GUI with the new sensor data and the sensor data is then stored in Amazon S3. The data is later input into Amazon Machine Learning to generate batch predictions.

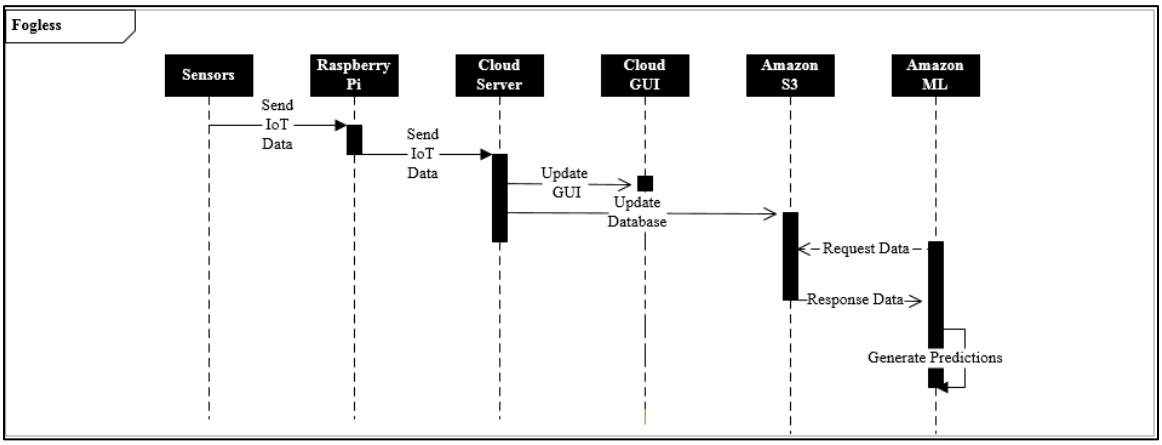


Figure 1: Sequence Diagram for Fogless Application

Figure 2 illustrates the high-level activities of the Fog-Enabled Application. All activities stay the same, except that there is a Fog layer between the Raspberry Pi and the Cloud server, which uses the sensor data to update a Fog GUI before forwarding the data to the Cloud server.

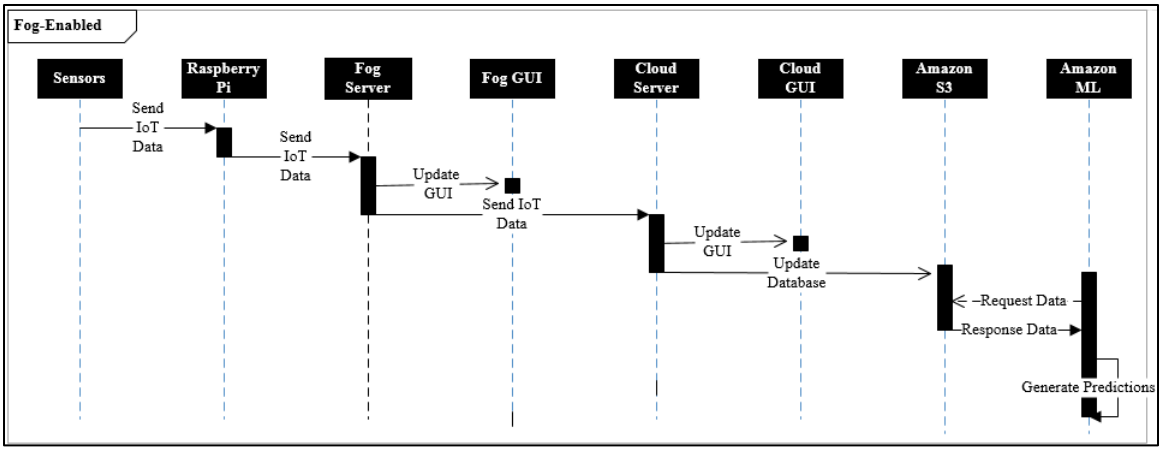


Figure 2: Sequence Diagram for Fog-Enabled Application

3.6 AWS Greengrass

Since one of the goals of this thesis is to provide researchers a better understanding of AWS Greengrass, a more detailed overview of GreenGrass and information regarding how Greengrass was utilized in the Fog-enabled application is provided in the following subsections. Researchers interested in implementing AWS Greengrass in their own applications are encouraged to first read about this project's lessons learned in section 6.2.1. The guide from Amazon, as it relates to AWS Greengrass, utilized for this project can be found on the Amazon Web Services website [Amazon Web Services18A]. Researchers and practitioners interested in recreating the results of this thesis can find detailed installation configuration and execution information for AWS Greengrass in Appendix A.

Figure 3 describes how devices interact along the IoT-Fog-Cloud spectrum. Devices using the IoT Device SDK can send data to the Greengrass Core device via the local

network. Then, the Greengrass Core can execute Lambda functions and interact with the Cloud, which is the model that was chosen for this thesis.

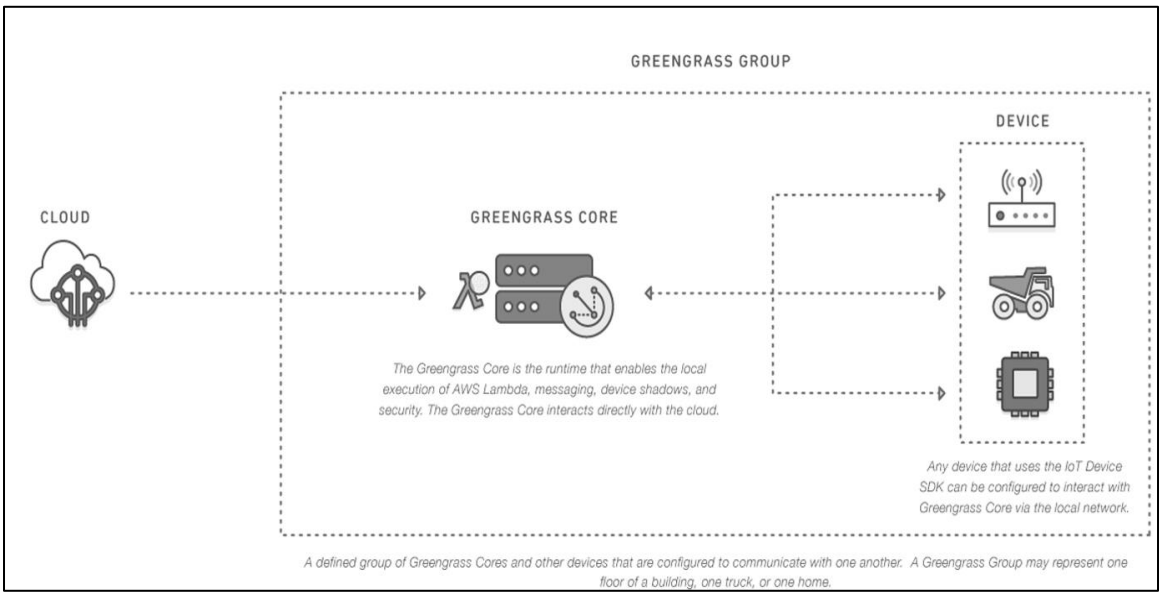


Figure 3: AWS Greengrass Interaction Model [AWS Greengrass17]

3.6.1 Lambda Functions on AWS Greengrass

Lambda functions enable users to deploy code without having to worry about provisioning or maintaining server software. In AWS Greengrass, Lambda functions are containerized along with any relevant code libraries and are run on the Greengrass device as either long-lived functions or on-demand functions.

In the AWS Greengrass paradigm, one must create a Lambda function in the AWS Management Console and deploy that function to the Greengrass core device via the Greengrass console. In this thesis, a Lambda function is used to capture IoT data and post

it to the Cloud server via a RESTful API. The Lambda function is shown in Figure 4. The requests library is used, so all dependencies for the requests library must be included with the Lambda function. An easy way to get all the dependencies for a given library on a local machine is to use pip, which is a Python package management system.

Figure 4: Lambda Functions on Greengrass Step 4

3.6.2 Device Creation in AWS IoT

To communicate between IoT devices, the Greengrass core and the Cloud, it's necessary to create devices in the Greengrass console. These devices represent physical objects and servers which will make up the IoT ecosystem in each application. A device needs to be created not only for the IoT devices, but for the Greengrass core device as well.

3.6.3 Subscription Configuration

AWS Greengrass uses a pub-sub model to send and receive data to and from devices. In this model, publishers publish data to topics, and subscribers subscribe to topics in which they're interested. Every message that is published to a topic is pushed immediately, with little or no queuing to all subscribers of that topic. A message broker lies between the publishers and subscribers and coordinates message flow between interested parties. AWS Greengrass provides a section in the Greengrass console for setting up subscriptions. In this thesis, there are 4 Subscriptions in the Greengrass console, as outlined in Figure 5.

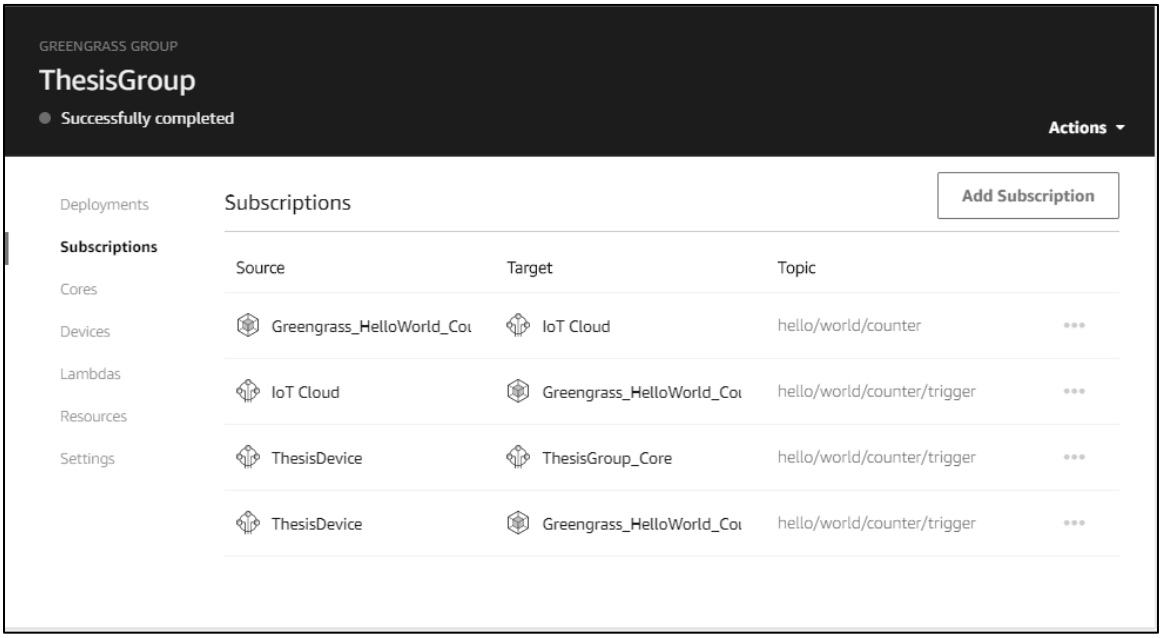


Figure 5: Thesis Greengrass Subscriptions

In this setup, the Raspberry Pi (ThesisDevice) publishes data to the Greengrass Core (ThesisGroup_Core) and the Lambda Function which sits on the Greengrass Core (Greengrass_HelloWorld_Counter). The top two subscriptions are for testing purposes only in the IoT Test console. The Test console provides an interface on AWS with which one can publish and subscribe to topics to test subscriptions from the Cloud. It's vitally important to get these subscriptions correct as there is currently no way in Greengrass to troubleshoot misconfigured subscriptions other than modifying the subscriptions, redeploying the Greengrass configuration to the Greengrass Core and checking with the IoT Test Console if the subscriptions are returning the correct data.

3.6.4 Configuration and Lambda Deployment

Once all the proper Subscriptions are in place, it is time to deploy the GreenGrass Group onto the Greengrass Core device. This step sends the configurations and Lambda functions to the device which is running the Greengrass software. This means that the Lambda functions are now running in the Fog instead of the Cloud!

3.6.5 Discovering the Greengrass Core

To send data from an IoT device to another device in the Greengrass Group, one must perform a Discovery. A Discovery is an initial request sent to the IoT Cloud which retrieves information about the Greengrass Core for the device to connect to the Core.

Once this initial Discovery is complete, the device is now able to communicate with the Greengrass Core directly, even when disconnected from the Cloud.

3.6.6 Sending Data from an IoT Device to the Greengrass Core

Once the Greengrass Discovery has been completed, it's now possible to create a custom script to publish data to the Greengrass Core device. The recommendation from the learnings of this research would be to start from the sample "BasicPubSub.py" script which is included in the Python IoT SDK provided by AWS. This script provides a general framework of a pub/sub application which seems to be a great way to learn to the IoT SDK syntax. The script itself contains in it comments of the parameters required to run the script, which are very similar to the script used to discover the GreenGrass Core. The example script used in this thesis can be found in Appendix A, in the section titled "Sending IoT Data to the Greengrass Core".

3.7 Application Demonstration

Screen captures of the finished GUIs can be found in Figure 6 through Figure 8. Demonstrations of the GUIs can be found at Nathan Wheeler's Thesis YouTube channel [Wheeler18]. Due to constraints in the project timeline and to provide a more focused approach, it was decided to limit the Single Coffee Shop GUI view to a single coffee cup and the Manager View to two coffee shops with a single coffee cup each.

Figure 6: Fog Single Coffee Shop GUI

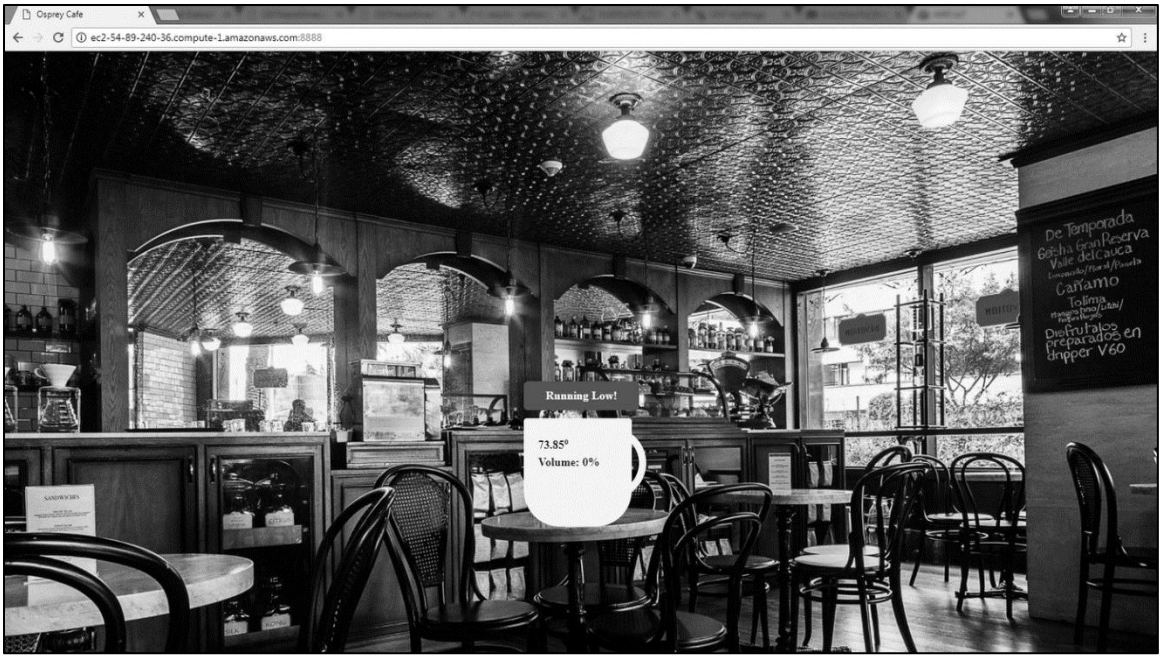


Figure 7: Cloud Single Coffee Shop GUI

Figure 8: Cloud Manager View GUI

3.8 Amazon Machine Learning

Since another of the goals of this thesis is to provide researchers a better understanding of Amazon Machine Learning, a more detailed overview of Amazon Machine Learning and information regarding how the API was utilized in this thesis is provided in the subsections to follow. The four-step process to produce predictions in Amazon Machine Learning API is as follows:

1. Create a New Datasource.
2. Create a New Machine Learning Model.
3. Evaluate the Machine Learning Model.
4. Generate Predictions with the Machine Learning Model.

This four-step process will be explained in greater detail in the following subsections. Researchers interested in implementing Amazon Machine Learning in their own applications are encouraged to first read about this project's lessons learned in section 6.2.2.

3.8.1 Creating a New Datasource

To begin creating a machine learning model in Amazon Machine Learning, a datasource must first be created. Datasource objects contain metadata about the input data. When a datasource is created, Amazon Machine Learning reads the input data, computes statistics which describe its attributes and stores these statistics along with a schema and other information as part of the datasource object. The input data which was used to create the datasource for this thesis consisted of about 6,000 records in Excel format. The records consisted of a timestamp, broken down into categorical attributes down to the millisecond, as well as the target, "winner" attribute, which represents the restaurant which was performing better during the given time.

When creating the datasource schema, Amazon ML tries to predict the data types for the input data. Amazon ML has 4 different data types:

Binary: Attributes that have a yes or no value.

Categorical: Attributes that take on a limited number of unique string values.

Numeric: Attributes that have a quantity as a value.

Text: Attributes that are a string of words. Amazon ML tokenizes this input and delimits by white space.

3.8.2 Creating a New Machine Learning Model

Amazon Machine Learning will infer a machine learning model type based on the provided datasource. In the case of this thesis, the MULTICLASS model type was suggested. There are 3 ML model types in Amazon ML:

Binary Classification Model: Predicts a binary outcome. This model uses the industry-standard learning algorithm logistic regression.

Multiclass Classification Model: Predicts one of more than two outcomes. This model uses the industry-standard learning algorithm multinomial logistic regression.

Regression Model: Predicts a numeric value. This model uses the industry-standard learning algorithm linear regression.

3.8.3 Evaluating the Machine Learning Model

In addition to the model type, training and evaluation settings are recommended. The default settings are that 70% of the datasource will be used to train the machine learning model and the remaining 30% of the datasource will be used to evaluate how well the model will work in the future. Keep in mind that if the distributions of the target values are massively different in the two datasets (training and evaluation), the resulting machine learning model will be evaluated as poor.

If it makes sense for the given datasource, the order of the records may need to be randomized to ensure a more even distribution of target values. In the case of this thesis, the records were randomized to solve for the issue of massively differing target values in the two datasets. The output from the model evaluation process is an average F1 score and a confusion matrix as shown in Figures 9 and 10.

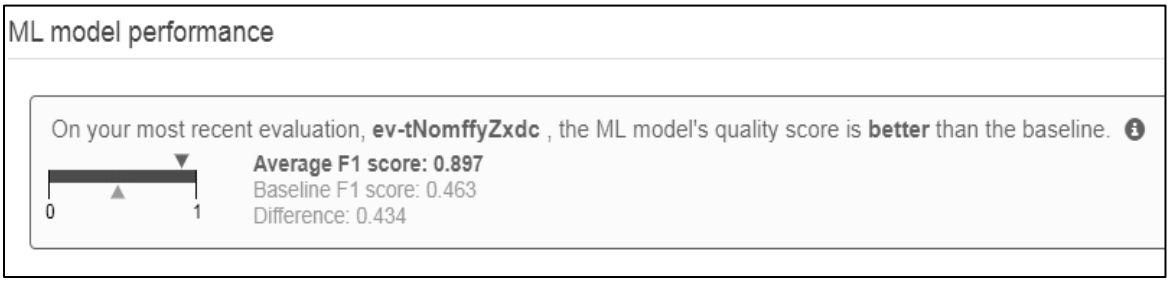


Figure 9: Updated ML Model Performance

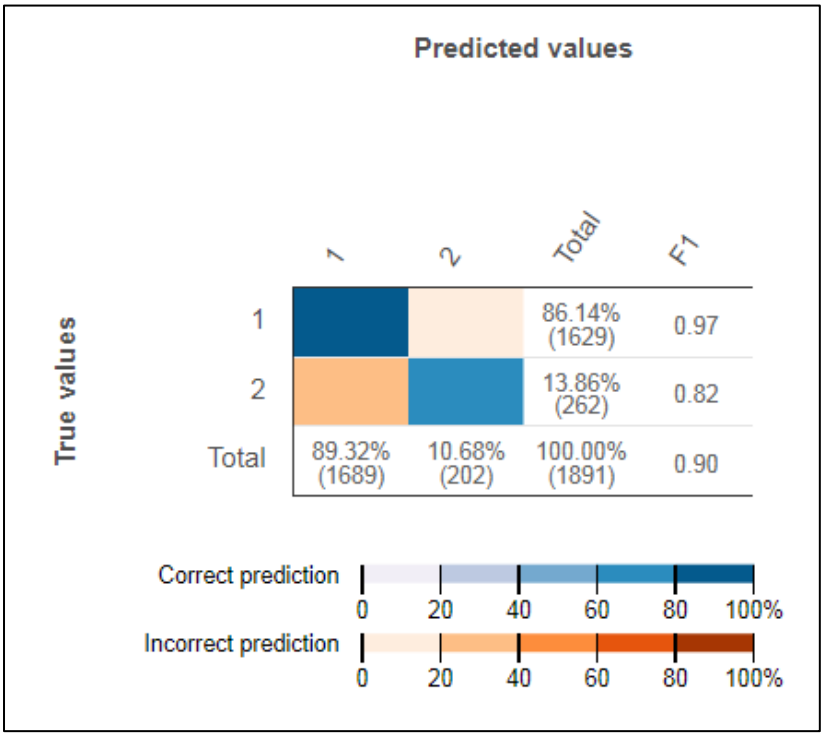
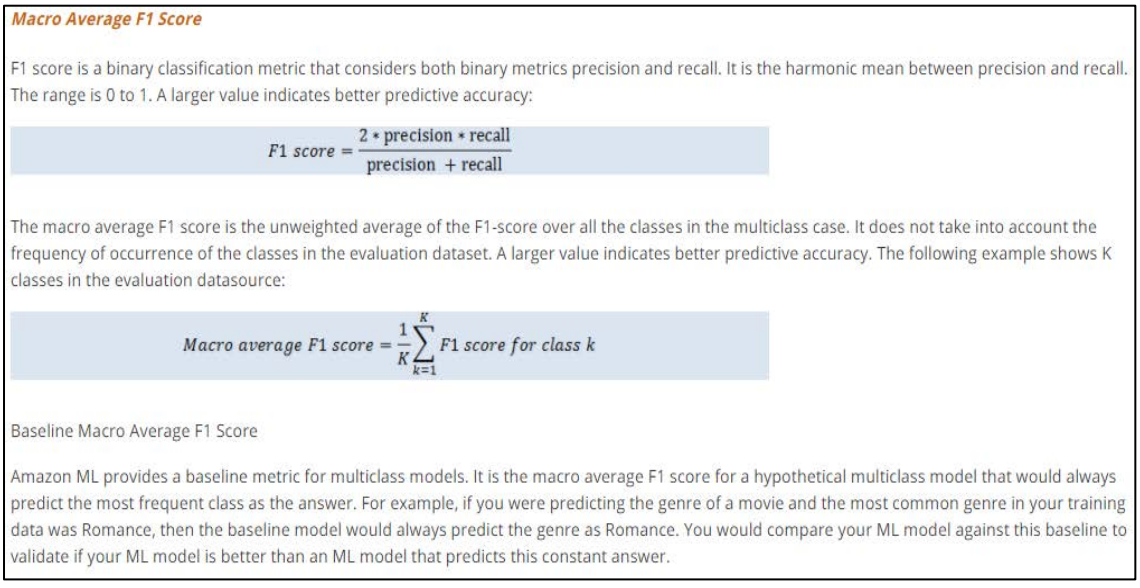


Figure 10: Amazon ML "Confusion Matrix"

The “Confusion Matrix” in Figure 10 can be interpreted as follows, adapted from Amazon Web Services [Amazon Web Services18C]:

Number of correct and incorrect predictions for each class: Each row in the confusion matrix corresponds to the metrics for one of the true classes. For example, the first row shows that for winners that are in the 1st coffee shop, the multiclass ML model gets the predictions right for over 80% of the cases. The model incorrectly predicts the winner as the second coffee shop for less than 20% of the cases.

Class-wise F1-score: The last column shows the F1-score for each of the classes. Figure 11 shows how Amazon Machine Learning calculates the F1 score.



Macro Average F1 Score

F1 score is a binary classification metric that considers both binary metrics precision and recall. It is the harmonic mean between precision and recall. The range is 0 to 1. A larger value indicates better predictive accuracy:

$$F1\ score = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

The macro average F1 score is the unweighted average of the F1-score over all the classes in the multiclass case. It does not take into account the frequency of occurrence of the classes in the evaluation dataset. A larger value indicates better predictive accuracy. The following example shows K classes in the evaluation datasource:

$$Macro\ average\ F1\ score = \frac{1}{K}\sum_{k=1}^{K} F1\ score\ for\ class\ k$$

Baseline Macro Average F1 Score

Amazon ML provides a baseline metric for multiclass models. It is the macro average F1 score for a hypothetical multiclass model that would always predict the most frequent class as the answer. For example, if you were predicting the genre of a movie and the most common genre in your training data was Romance, then the baseline model would always predict the genre as Romance. You would compare your ML model against this baseline to validate if your ML model is better than an ML model that predicts this constant answer.

Figure 11: Description of Amazon's F1 Value [Amazon Web Services18C]

True class-frequencies in the evaluation data: The second to last column shows that in the evaluation dataset, 86.14% of the observations in the evaluation data is the 1st coffee shop as the winner, 13.86% is the 2nd coffee shop as the winner.

Predicted class-frequencies for the evaluation data: The last row shows the frequency of each class in the predictions. 89.32% of the observations are predicted as coffee shop 1, 10.68% is predicted as coffee shop #2.

3.8.4 Generating Predictions with the Machine Learning Model

Amazon Machine Learning provides the ability to perform batch predictions and real-time predictions with created machine learning models. For the purposes of this thesis, since the extra speed of the real-time API was not needed, the less costly batch prediction option was chosen. The data for which batch predictions need to be made must be loaded into S3 and used to feed into the machine learning model. A sample of the prediction data used is shown in Figure 12. These prediction data will tell which coffee shop will perform better at 8 A.M., 12 P.M. and 5 P.M. of the next business day.

	A	B	C	D	E	F	G
1	Month	Day	Year	Hour	Minute	Second	Millisecond
2	4	2	2018	8	0	0	0
3	4	2	2018	12	0	0	0
4	4	2	2018	17	0	0	0

Figure 12: Generating Batch Predictions Part 5

The results of the batch predictions look like Figure 13, with a row for each record that was input with the prediction datasource. All the columns add up to 100%. The numbers produced from Amazon Machine Learning are in scientific notation.

	A	B
1	2	1
2	3.35E-03	9.97E-01
3	3.35E-03	9.97E-01
4	3.35E-03	9.97E-01

Figure 13: Batch Prediction Results

These results indicate that there's a 99.67% chance that, with the given input data, restaurant #1 will be the winner. There's a .0033% chance that, with the given input data, restaurant #2 will be the winner. Since the coffee shop refill rates were set ahead of time, one can judge these predictions as accurate since coffee shop # 1 was the faster refilling coffee shop and had a lower average refill rate at the tail end of the data set.

To create the data sources, machine learning model, evaluation and predictions from the ~6,000 records, the total cost was about $1 USD.

Chapter 4

METRICS AND SPECIFICATIONS

4.1 Metrics

The metrics which were used to measure performance of the two applications are one-way transfer time and round-trip transfer time. When compared, these metrics capture the difference in response time of the two applications, allowing us insight into the benefits of utilizing Fog in a 3-tiered architecture. The metrics are explained further in the following subsections.

4.1.1 One-way Transfer Time (Milliseconds)

One-way transfer time includes the time it took for the source device to timestamp the request data and for the data to be timestamped again at the destination. The code is arranged in such a way that no one scenario contains a difference in code execution time.

4.1.2 Round-Trip Transfer Time (Seconds)

Round-Trip Transfer Time consists of the time it took the IoT device to receive a response and continue executing. We measured the total time taken for 31 synchronous requests to finish executing to provide a significant sample size.

The following statistics were calculated based on the transfer time data collected:

Minimum: The minimum value.

Q1: The cutoff value for the first Quartile.

Median: The middle value.

Q3: The cutoff value for the 3rd Quartile

Maximum: The maximum value.

Mean: The average value.

IQR: Interquartile range is the range from Q1 to Q3 (Q3-Q1).

*High Outlier = > IQR *1.5 + Q3*: The point at which a value is considered a high outlier.

*Low Outlier = < Q1- IQR *1.5*: The point at which a value is considered a low outlier.

Range: The maximum minus the minimum value, or the range of values.

4.2 Specifications

To help replicate the experiments outlined in this thesis, a hardware, software and network specification are provided. The specifications in the following subsections

outline the components used at each of these three specification layers within the applications.

4.2.1 Hardware Specification

The hardware specification outlines the hardware components which are used in the applications to a level of detail sufficient to recreate the experiments. The hardware specification is further divided for organizational purposes into Fog-enabled and Fogless hardware specifications.

4.2.1.1 Fog-Enabled Application

The hardware specification of the Fog-enabled application outlines the hardware components used at the three different levels of the application: IoT, Fog and Cloud. The hardware used at each level is outlined in the following subsections.

4.2.1.1.1 IoT-Level

Raspberry Pi Model 3 B - 1.2 GHZ quad-core ARM, 1 GB RAM 10/100 MBPS Ethernet, 802.11n Wireless LAN.

4.2.1.1.2 Fog-Level

Virtual Machine with 1 GB RAM and up to 3.3GHz Clock Speed

Technicolor Tc8305c WIFI Cable Modem Router

4.2.1.1.3 Cloud-Level

AWS T2 micro instance with 1 GB RAM and up to 3.3GHz Clock Speed.

4.2.1.2 Fogless Application

The hardware specification of the Fogless application outlines the hardware components used at the two different levels of the application: IoT and Cloud. Since this thesis aims maintain a like to like comparison, the same hardware is used at the IoT-level and the Cloud-level for both applications. Therefore, the Fogless application hardware specification is the same as the Fog-enabled application hardware specification with the subtraction of the Fog-level hardware.

4.2.2 Network Specification

Fogless (4g LTE): 19.9 Mbps download/2.56 upload/66ms ping

Fog-Enabled (Comcast Xfinity): 120.87 Mbps download/12.05 Mbps upload/12ms ping

4.2.3 Software Specification

The software specification outlines the software components used at the three different levels of the two applications: IoT, Fog and Cloud. Since the software used at a given level was kept common across both applications, the software used at each level is outlined in the following subsections without regard to either application.

4.2.3.1 IoT-Level

Operating System: Raspbian Jesse; AWS IoT Device SDK for Python;

4.2.3.2 Fog-Level

Operating System: Ubuntu 16.04 LTS; AWS Greengrass Core; AWS Greengrass SDK;
Server: Node.js 8.0.0 with Express.js; HTML5; CSS3; jQuery 3.3.1;

4.2.3.3 Cloud-Level

Operating System: Ubuntu 16.04 LTS;
Server: Node.js 8.0.0 with Express.js; HTML5; CSS3; jQuery 3.3.1; Amazon S3;
Amazon Machine Learning;

Chapter 5

TEST METHODOLOGY

5.1 Application Requests

Requests to the Fog server utilized the AWS IoT SDK's publish function. Using this function allows the message broker from AWS Greengrass located on the fog server to trigger the Lambda function which is subscribed to the topic to which data was just published. This interaction is the main difference between a pub-sub communication model and a typical request and response client-server model. Requests to the Cloud server utilized synchronous POST requests from the Python Requests library.

5.2 Transfer Time Study Measurements

Both one-way and round-trip transfer time were measured 3 times per day over the course of two days. During each measurement, 31 requests were performed to ensure a significant sample size of requests. For one-way, the transfer times were averaged. For round-trip, the transfer times for the 31 requests were summed and compared between the two applications. The one-way study was performed for 3 scenarios: Fog-Enabled (Including Fog in the IoT-Fog-Cloud architecture), Fogless (IoT and Cloud only) and Fog-only (IoT-Fog only). The round-trip study was performed for Fog-Enabled and

Fogless. After plotting the data points listed in the Metrics and Specification section, the data are analyzed.

Chapter 6

RESULTS AND ANALYSIS

6.1 Study on Application Response Times

In Table 1 through Table 4, transfer time data and statistics from the response times studies are presented. Then, the results and analyses will start with the one-way transfer time boxplot provided in Figure 14.

Time in Milliseconds			
	Fog-Enabled Transfer Time	Fogless Transfer Time	Fog-Only Transfer Time
Day 1 Afternoon	904.35	1602.21	690.05
Day 1 Evening	921.32	867.76	602.5
Day 2 Afternoon	907.20	886.93	631.57
Day 2 Evening	943.68	1322.66	583.37
Day 3 Afternoon	887.42	1282.29	633.4
Day 3 Evening	921.33	997.05	590.65

Table 1: One-Way Transfer Time in Milliseconds

One-Way Transfer Time Statistics			
	Fog-Enabled	Fogless	Fog-Only
Minimum	887.42	867.76	583.37
Q1	905.06	914.46	593.61
Median	914.26	1139.67	617.04
Q3	921.33	1312.57	632.94
Maximum	943.68	1602.21	690.05
IQR	16.27	398.11	39.33
High Outlier = > (IQR * 1.5 + Q3)	945.74	1909.74	691.94
Low Outlier = < (Q1 - IQR * 1.5)	880.66	317.3	534.62
Mean	914.22	1159.82	621.92
Range	56.26	734.45	106.68

Table 2: One-Way Transfer Time Statistics

Time in Seconds		
	Fog-Enabled Transfer Time	Fogless Transfer Time
Day 1 Afternoon	27	78.72
Day 1 Evening	26.8	30.7
Day 2 Afternoon	26.76	28.84
Day 2 Evening	26.85	42.53
Day 3 Afternoon	26.85	43.03
Day 3 Evening	26.69	33.18

Table 3: Round-Trip Transfer Time in Seconds

Round-Trip Transfer Time Statistics		
	Fog-Enabled	Fogless
Minimum	26.69	28.84
Q1	26.77	31.32
Median	26.83	37.86
Q3	26.85	42.91
Maximum	27	78.72
IQR	0.08	11.59
High Outlier = > (IQR * 1.5 + Q3)	26.97	60.28
Low Outlier = < (Q1 - IQR * 1.5)	26.65	13.94
Mean	26.83	42.83
Range	0.31	49.88

Table 4: Round-Trip Transfer Time Statistics

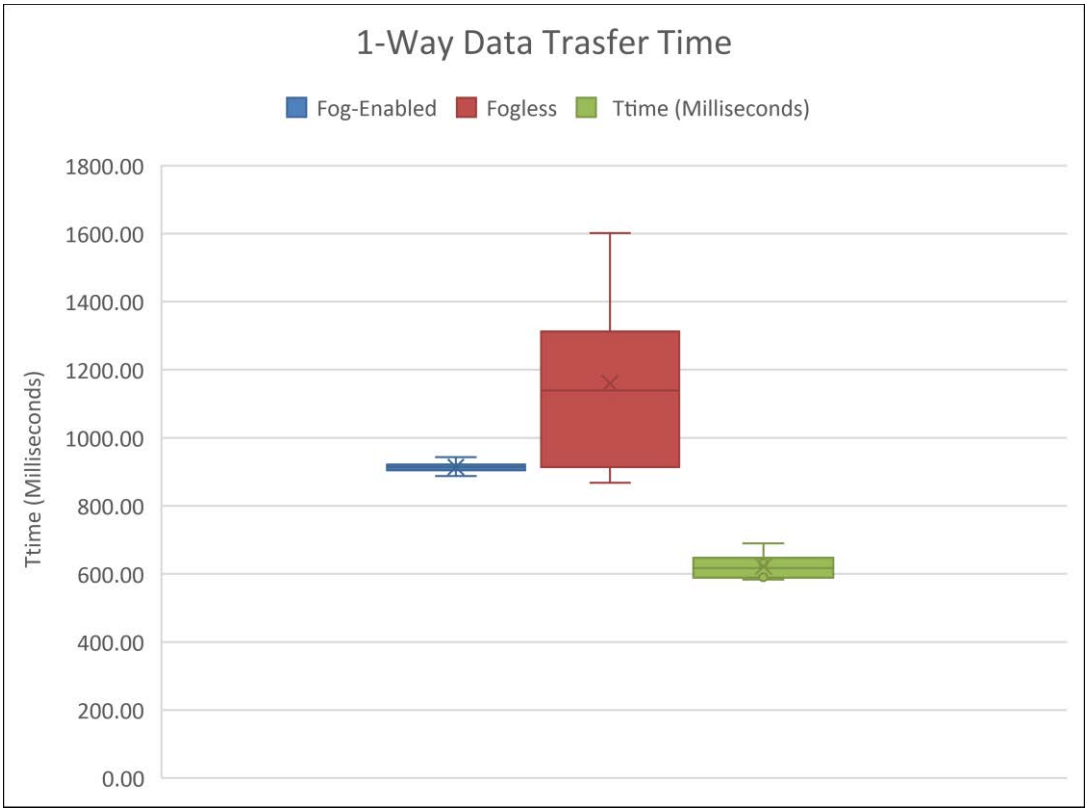


Figure 14: Box Plot for 1-Way Data Transfer Statistics

As can be gathered from the 1-Way Data Transfer Time box plot in Figure 14, the range of response times is much higher for the Fogless application, a 678-millisecond difference. In addition, the maximum response time for the Fogless application is much higher than either the Fog-Enabled or Fog-Only maximums. The lower ranges for the Fog-Enabled and Fog-Only solutions indicate much better network stability.

Since the minimum of the Fogless application is lower than minimum of the Fog-enabled application, we know that the problem is not one of raw speed. However, it's clear that the Fog-only solution has both much lower values than the Fogless application and a much lower range of values as well. This fact indicates that the Fog is indeed more apt for applications needing a quicker response time and more stable responses. Furthermore, the Fog-Enabled application also shared a much lower range of values and much lower values on average than the Fogless application. This face indicates that using a Fog device as a Platform for IoT and Cloud applications is indeed beneficial to the overall stability and speed of the application.

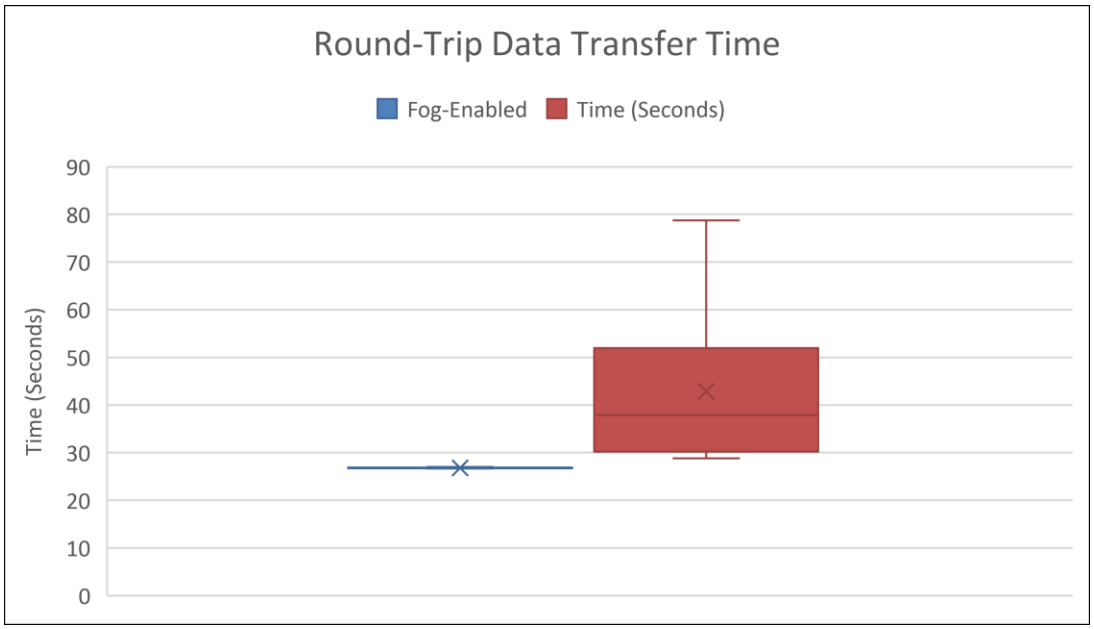


Figure 15: Box Plot for Round-Trip Transfer Time Statistics

The box plot from Figure 15 regarding round-trip transfer time tells the same story as the one-way transfer time study, though more convincingly so. The range of the Fog-Enabled response time is so small that it isn't legible on the box plot. Furthermore, the maximum total response time for the Fog-Enabled application never goes above the minimum response time of the Fogless application. Both studies bode well for the utility of utilizing Fog in a 3-tier architecture with IoT and Cloud.

6.2 Lessons Learned

The Lessons Learned section enumerates the learnings which have been gleamed from application development with cutting edge technologies within this thesis. This section is split among the AWS Greengrass software package and the Amazon Machine Learning

Cloud API. Some of the lessons contained herein for AWS Greengrass were obtained directly from Amazon Solutions Architects who developed the software platform.

6.2.1 AWS Greengrass

1. The Greengrass guide for Linux doesn't indicate add a user and group must be added as follows before installing greengrass core:
   sudo adduser --system ggc_user
   sudo addgroup --system ggc_group
2. Instructions do not mention that the user must configure the config.json file which was downloaded while setting up the Greengrass Group.
3. If the basicDiscovery.py is producing a 403: Forbidden response, it's possible the Certificate Authority was not activated when creating the device. This issue was solved by deleting and recreating the device in the Greengrass Console.
4. During this research, deployment failures were mainly caused by Subscriptions pointing to devices that no longer existed.
5. Creating an Alias for a Lambda function makes it so only the pointer to the Lambda version number needs to be changed when publishing a new version of the function. Without Aliases, all Subscriptions must be re-created after each newly published version.
6. When updating the version to which a Lambda Alias points, exit out of that screen before trying to deploy the Greengrass Group, or the deployment may hang.

6.2.2 Amazon Machine Learning

1. Certain types of data perform better when used to evaluate a machine learning model. For example, in this case, since the last 30% of the data contained the same target attribute value for 100% of the values, the model will not be as accurate in predicting the value which was not present in the evaluation dataset.
2. To predict a target attribute, Amazon Machine Learning must have a large sample size of data with that attribute present as well as all attributes one will be using to predict that target attribute. For example, if one wants to predict the stock prices based on past interest rates, one must provide many records of past stock prices and together with the corresponding interest rate.

Chapter 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This thesis has driven the development and analysis of two applications whose architectures are of both business and research value. In providing studies which analyze the differences in response times between these two applications, we've shown that an IoT-Fog-Cloud architecture has advantages over an IoT-Cloud architecture. This research has demonstrated a real-world use-case of a coffee shop in which Fog Computing would be of use to a business by performing data processing on the Edge of the network, which is not possible in the Cloud. In our example use-case Fog also served as a liaison between the IoT and the Cloud, ensuring stable and fast response times. Results suggest that Fog-enabled applications have a much lower range of response times as well as lower response times overall. These results suggest Fog-enabled solutions are suitable for applications which require network stability and reliably lower latency.

Furthermore, the argument has been strengthened by the integration and illustration of a high-performance application, machine learning, which cannot be effectively performed on the Fog, so Cloud is a necessary and complementary component of the 3-tier architecture. In addition, this thesis has documented the user experiences of the relatively unstudied and new AWS products, Amazon Greengrass and Amazon Machine Learning,

to provide the research and practitioner communities an improved understanding of these products so that superior research and products can be created in the future.

7.2 Future work

In future projects Fog can be utilized to cache data on the Edge, further reducing response times in IoT-Fog-Cloud architectures. Fog can also be utilized with Software Defined Networking, reducing response times even further and providing Quality of Service Guarantees. In more security-conscious applications, Fog can also provide security by filtering out sensitive data at the Edge before sending it through to the WAN. There's an opportunity for a vendor or business to create a scalable Fog solution which is easy to plug and play into common business and research infrastructures in which Fog is of benefit.

REFERENCES

Print Publications:

[Chiang16]
Chiang, M., & Zhang, T. (2016). “Fog and IoT: An Overview of Research Opportunities”, IEEE Internet of Things Journal, 3(6), 854-864.

[Chieochan17]
Chieochan, O., Saokaew, A., & Boonchieng, E. (2017). “IOT for smart farm: A case study of the Lingzhi mushroom farm at Maejo University” 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE).

[Dutta17]
Dutta, J., & Roy, S. (2017). “IoT-Fog-Cloud based architecture for smart city: Prototype of a smart building”, 2017 7th International Conference on Cloud Computing, Data Science & Engineering – Confluence.

[Giang15]
Giang, N. K., Blackstock, M., Lea, R., & Leung, V. C. (2015). “Developing IoT applications in the Fog: A Distributed Dataflow approach”, 2015 5th International Conference on the Internet of Things (IOT).

[Husni16]
Husni, E., Hertantyo, G. B., Wicaksono, D. W., Hasibuan, F. C., Rahayu, A. U., & Triawan, M. A. (2016). “Applied Internet of Things (IoT): Car monitoring system using IBM BlueMix”, 2016 International Seminar on Intelligent Technology and Its Applications (ISITIA).

Electronic Sources:

[Amazon18]
Amazon.com, “Samples for AWS Greengrass”, https://github.com/aws-samples/aws-greengrass-samples

[Amazon Web Services18A]
Amazon Web Services, Inc., “Getting Started with AWS Greengrass”, https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-gs.html

[Amazon Web Services18B]
Amazon Web Services, Inc., “Splitting Your Data”,
https://docs.aws.amazon.com/machine-learning/latest/dg/splitting-types.html

[Amazon Web Services18C]
Amazon Web Services, Inc., “Multiclass Model Insights”,
https://docs.aws.amazon.com/machine-learning/latest/dg/multiclass-model-insights.html?icmpid=docs_machinelearning_console

[AWS Greengrass17]
AWS Greengrass – “Embedded Lambda Compute in Connected Devices - Amazon Web Services”, https://aws.amazon.com/greengrass/

[Canonical Ltd.18]
Canonical Ltd., “Home”, https://www.ubuntu.com/

[Cisco15]
Cisco and/or its affiliates, “Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are”,
https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf

[Gartner17]
Gartner Inc., “Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016”, https://www.gartner.com/newsroom/id/3598917

[IDC17]
International Data Corporation, “Worldwide Spending on the Internet of Things Forecast to Reach Nearly $1.4 Trillion in 2021, According to New IDC Spending Guide”,
https://www.idc.com/getdoc.jsp?containerId=prUS42799917

[Node.js Foundation18A]
Node.js Foundation, “Home”, https://nodejs.org/en/

[Node.js Foundation18B]
Node.js Foundation, “Home”, https://expressjs.com/

[Pi My Life Up16]
Pi My Life Up, “Raspberry Pi Temperature Sensor: Build a DS18B20 Circuit”,
https://www.youtube.com/watch?v=sX9KCK2Z-I0&t=112s
[Raspberry Pi Foundation18]
Raspberry Pi Foundation, “Welcome to Raspbian”, https://www.raspbian.org/

[Wheeler18]
Nathan Wheeler, “Temperature Sensor Demo”, “Manager View”, “Iot to Fog”, “Iot to Cloud”,
https://www.youtube.com/channel/UCog3ekQJHCLsKk_xOTXiU4Q?view_as=subscriber

[Zhuo Chen16]
Zhuo Chen, Kiryong Ha, Hu, Wenlu, Richter, Wolfgang, Pillai, Padmanabhan, & Satyanarayanan, Mahadev (2016). “Towards Wearable Cognitive Assistance”, Carnegie Mellon University and Intel Labs, http://elijah.cs.cmu.edu/DOCS/ha-mobisys2014.pdf

APPENDIX A

AWS Greengrass Installation, Configuration and Execution

Environment Setup for Greengrass

Since the Fog device used in this research was a virtual machine running on the Oracle VM Virtualbox software, the “Setting Up Other Devices” section in Module 1 of the Greengrass guide was followed.

Step 1: Ensure the device has the necessary dependencies by downloading and extracting the Greengrass dependency checker from the AWS GitHub Repository:

Run the following commands:

git clone https://github.com/aws-samples/aws-greengrass-samples.git

cd greengrass-dependency-checker-GGCv1.5.0

sudo ./check_ggc_dependencies

Step 2: Install all dependencies called out by the dependency checker script.

Installing the Greengrass Core Software

Step 1: Sign into the AWS Management Console and open the AWS IoT console, then choose “Greengrass”.

Step 2: On the Welcome to AWS Greengrass page, choose “Get Started”.

Step 3: Create an AWS Greengrass group using the easy creation option.

Step 4: Type in a name for the group and choose “Next”.

Step 5: Use the default name for the core and choose “Next”.

Step 6: Run a scripted easy Group creation and choose “Create Group and Core”.

Step 7: Download the core’s security resources and the AWS Greengrass Core software. In this case, this is the x86_64 Ubuntu 14.04-16.04 Linux option.

Step 8: Choose “Finish and View the New Greengrass Group”.

Start AWS Greengrass on the Core Device

Step 1: Transfer the core’s security resources and the AWS Greengrass Core software downloaded in the prior section to the core device.

Step 2: Navigate to the path where the two compressed files from step 1 are saved and run the below commands to decompress the files:

sudo tar -xzvf greengrass-***OS-architecture***-1.5.0.tar.gz -C /

sudo tar -xzvf ***GUID***-setup.tar.gz -C /greengrass

Among other things, the first command creates the /greengrass directory in the root folder of the AWS Greengrass core device (via the -C / argument). The second command copies the certificates into the /greengrass/certs folder and the config.json file into the /greengrass/config folder (via the -C /greengrass argument).

Step 3: Install the Symantec VeriSign root CA onto the device. This certificate enables the device to communicate with AWS IoT using the MQTT messaging protocol over

TLS. Make sure the AWS Greengrass core device is connected to the internet, then run the following commands (note that -O is the capital letter O):

1. cd /greengrass/certs/

2. sudo wget -O root.ca.pem http://www.symantec.com/content/en/us/enterprise/verisign/roots/VeriSign-Class%203-Public-Primary-Certification-Authority-G5.pem

Confirm that the root.ca.pem file isn't empty by executing the following command: cat root.ca.pem.

Step 4: Use the following commands to start AWS Greengrass:

cd /greengrass/ggc/core/

sudo ./greengrassd start

The output should look like Figure 16.

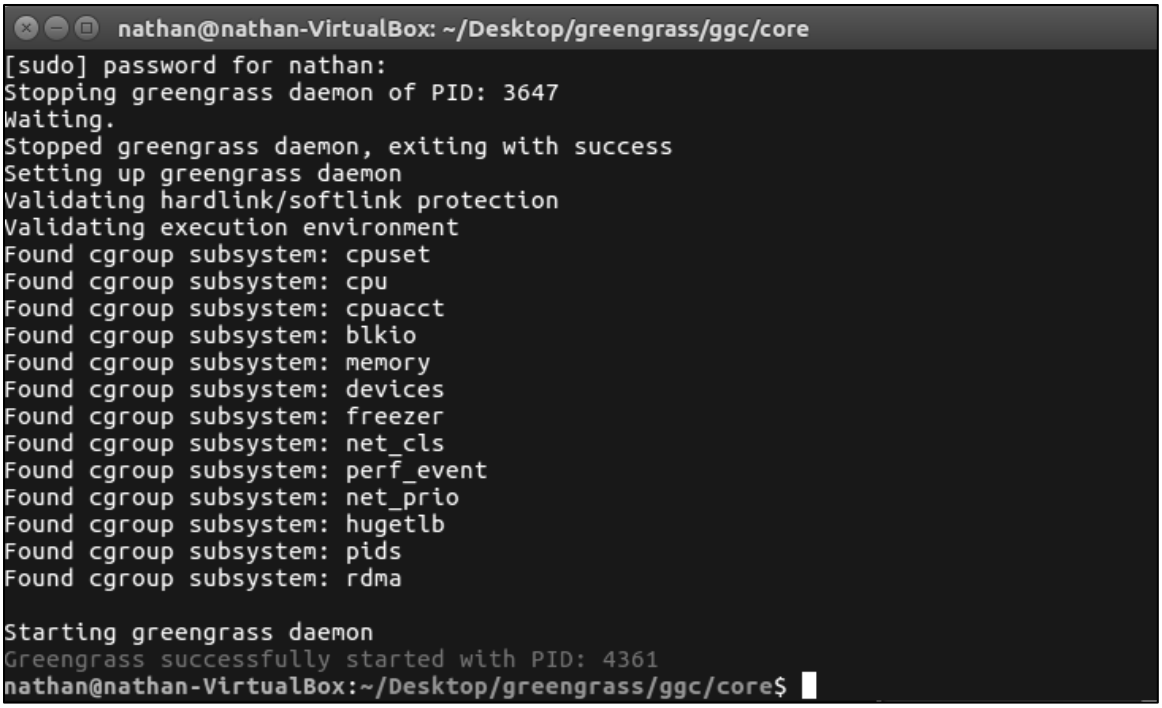


Figure 16: Greengrass Start Output

Step 5: Use the following command in to ensure that the AWS Greengrass core daemon is functioning:

ps aux | grep ***PID-number***

The output should look like the output in Figure 17.

Figure 17: Greengrass Core Daemon Check Output

Step 6: Modify the greegrass/config/config.json file with the specific AWS values as in Figure 18.

Figure 18: Greengrass config.json File

Note: These are not actual values, but the format should look similar.

Without this step, the Greengrass core device will not be able to properly connect to AWS IoT in the Cloud.

Lambda Functions on AWS Greengrass

Below are the steps which were taken to create the Lambda function used in this project:

Step 1: Download the Greengrass Software Development Kit (SDK) from the Software section of the AWS IoT menu. This SDK allows one to use functions in one's code specific to the AWS Greengrass software.

Step 2: Decompress the downloaded SDK with the below command:

sudo tar -xzf greengrass-core-python-sdk-1.1.0.tar.gz

Step 3: Navigate to aws_greengrass_core_sdk/sdk/python_sdk_1_0_0 where you will find the 3 folders which are necessary to include with any Python Greengrass Lambda function: greengrass_common, greengrass_ipc_python_sdk and greengrasssdk.

Step 4: To follow along with this research project, create a Greengrass_HelloWorld_Counter.py file as in Figure 19.

Figure 19: Lambda Functions on Greengrass Step 4

As is shown Figure 19, the requests library is used, so all dependencies for the requests library must be included with the Lambda function. An easy way to get all the

dependencies for a given library on a local machine is to use pip. Take the following example:

pip install requests

Note: Ensure that Python and Pip are installed for the above command to work. To install these on Ubuntu 16.04, use the following command:

sudo apt-get install python-pip

Use the following command to verify the installation: pip -V

Step 5: Once all dependencies are in the same folder, zip the folder.

Step 6: Open the Lambda Console and create a new Lambda function.

Step 7: Upload the zip file with the Lambda function and dependencies and choose Save.

Step 8: Publish the new version.

Step 9: Create an Alias for ease of development.

Note: Aliases allow users a pointer to a specific Lambda function, so users don't have to change references to the functions in their code. Point to version 1 for the first published version.

Device Creation in AWS IoT

Below are the steps which were taken to create the ThesisDevice and the ThesisGroup_Core device:

Step 1: In the devices tab in the Greengrass console, select “Add Device”.

Step 2: Choose “Create New Device”.

Step 3: Choose a name for the device.

Step 4: Choose “Use Defaults” under “1-Click”.

Step 5: Download the generated security credentials for the device, including the root Certificate Authority from VeriSign and choose Finish. Note that these credentials will be used to identify the device when making requests to Amazon IoT and Greengrass core, so it’s important to store the credentials securely.

Subscription Configuration

The process of setting up a subscription is outlined below:

Step 1: In the Subscription section in the Greengrass console, click “Add Subscription”.

Step 2: In the Select a source field, choose the publishing Service, Device or Lambda.

Step 3: In the Select a target field, choose the subscribing Service, Device or Lambda and click “Next”.

Step 4: Add a topic to which the publisher can publish, and the subscriber can subscribe and click “Next”, then “Finish”.

Lambda Deployment

To deploy the group, click on the Actions menu in the top-right of the Greengrass console and click “Deploy”. If there are any errors in the deployment, one may click the value in the Status column, find their deployment in the Deployments and Errors section, and once again click the value of Status column.

Discovering the Greengrass Core

To perform a Discovery, follow the steps outlined below:

Step 1: Install the AWS IoT Device SDK for Python on the IoT Device.

```
cd ~
git clone https://github.com/aws/aws-iot-device-sdk-python.git
cd aws-iot-device-sdk-python
sudo python setup.py install
```

Step 2: Navigate to the SDK’s samples folder, the greengrass folder, then copy the basicDiscovery.py file to the folder containing the IoT Device Certificates downloaded during the Device Set up.

Step 3: Ensure connectivity between the IoT Device and the Greengrass Core Device. In this case, the command, ***hostname -I*** was run on the Greengrass Core device to find the IP address, then the IP address was pinged from the Raspberry Pi (IoT Device).

Specific to hosting the Greengrass Core on a Virtualbox VM, ensure that the virtual machine is set to “Bridged Adapter” mode in the network settings. This allows the hypervisor to intercept requests and inject data into it, effectively creating a new network interface in software that enables routing or messages to and from the virtual machine.

Step 4: From the AWS IoT Menu, choose “Greengrass”, “Groups”, and then choose the group. Choose “Settings”, then select “Manually manage connection information”. Then choose “View Cores for specific endpoint information”. Choose the core, then “Connectivity”. Choose “Edit” and ensure there’s only one Endpoint value which must be the IP address of the AWS Greengrass core device (for port 8883). Choose “Update”. This ensures that the basicDiscovery.py script will connect to the correct AWS Greengrass core device IP address.

Step 5: Navigate to the folder holding the device certificates and run the following command (Note that the AWS_IOT_ENDPOINT parameter can be found in the Settings section of the AWS IoT Console):

python basicDiscovery.py --endpoint AWS_IOT_ENDPOINT --rootCA root-ca-cert.pem --cert publisher.cert.pem --key publisher.private.key --thingName ThesisDevice --topic 'hello/world/counter' --mode publish --message 'Hello, World! Sent from Raspberry Pi'

Sending IoT Data to the Greengrass Core

For the coffee shop use-case, an example script is given in Figure 20 through Figure 22, which is used to publish sensor data to a Lambda function on the Greengrass Core device. In addition to the actual temperature sensor, there is code to simulate a weight sensor, a customer drinking coffee at a semi-random speed and a server refilling drinks at a semi-random speed.

```
# Publish to the same topic in a loop forever. #Global Vars
loopCount = 0
cup_volume = 100
cup_time = datetime.now()
refill_count = 0
previous_message = {}
average_refill_time = 0
running_low = 0
customer_drinking_rate = random.randint(10,26)
```

Figure 20: Example Pub/Sub Script Part 1

```
waiter_refill_time = random.randint(5, 11)
while True:
    if args.mode == 'both' or args.mode == 'publish':
        message = {}
        temp = {}
        now = datetime.now()
        now_as_string = now.strftime('%m:%d:%Y:%H:%M:%S:%f')
        message['Volume'] = getVolumeData(now, customer_drinking_rate)
        temp['Temperature'] = read_temp()
        message['Celcius'] = temp['Temperature'][0]
        message['Farenheit'] = temp['Temperature'][1]
        message['DateTime'] = now_as_string#'%d:%d:%d:%d:%d:%d:%d' %
(now.month, now.day, now.year, now.hour, now.minute, now.second,
now.microsecond)
        message['LoopCount'] = loopCount
        message['RaspberryPiId'] = 1
        message['RefillCount'] = refill_count
        message['AvgRefillTime'] = average_refill_time
        message['RunningLow'] = running_low
        # End Message
        #Need a Refill!
        if not previous_message:
            print "This is the first Message. No Comparison can be done."
            if message['Volume']<=20 and message['RunningLow'] == 0:
                message['RunningLow'] = message['DateTime']
        elif message['Volume']<=20 and message['RunningLow'] == 0:
            message['RunningLow'] = message['DateTime']
        #Let the server wait for a bit
        if message['RunningLow'] != 0:
            running_low_datetime =
datetime.strptime(message['RunningLow'], "%m:%d:%Y:%H:%M:%S:%f")
            difference_now_and_running_low = now - running_low_datetime
            print(difference_now_and_running_low)
            if difference_now_and_running_low.seconds >=
waiter_refill_time:
                print(waiter_refill_time)
                print("Waiter is Refilling!")
                waiter_refill_actual_time = float('%d.%d' %
(difference_now_and_running_low.seconds,
difference_now_and_running_low.microseconds))
                wrat_rounded = round(waiter_refill_actual_time, 2)
                print wrat_rounded
        #Now the server can refill the drink
                cup_volume = 100
                cup_time = now
```

Figure 21: Example Pub/Sub Script Part 2

```
                refill_count+= 1
                message['RefillCount']+=1
                message['RunningLow']= 0
                customer_drinking_rate = random.randint(1,26)
                waiter_refill_time = random.randint(7, 13)
                print ('((%f * %d) + %f)/%d' %
(previous_message['AvgRefillTime'],previous_message['RefillCount'],
wrat_rounded, message['RefillCount']))
                new_avg = round(((previous_message['AvgRefillTime'] *
previous_message['RefillCount']) + wrat_rounded)/message['RefillCount'],2)
                message['AvgRefillTime'] = new_avg
                average_refill_time = new_avg
                print(message['AvgRefillTime'])
        #Publish Message to GreenGrass
        messageJson = json.dumps(message)
        myAWSIoTMQTTClient.publish(topic, messageJson, 0)
        if args.mode == 'publish':
            print('Published topic %s: %s\n' % (topic, messageJson))
        loopCount += 1
        cupTime = now
        previous_message = message
        running_low = message['RunningLow']
```

Figure 22: Example Pub/Sub Script Part 3

VITA

Nathan Wheeler earned his MBA from the University of North Florida. He has worked in the Global Digital and Cloud Technology department of Citibank North America as a Lead Application Developer and a Solutions Architect on high-profile projects like Citi Apple Pay, Citi Quick Lock and Citi Credit Cards Marketing. He is currently pursuing an MS degree in Computing and Information Sciences with a Computer Science concentration at the University of North Florida. He plans to go on to do a PhD with a focus on IoT, Cloud Computing and Machine Learning.