Cal Poly Computer Engineering Senior Project

Teaching the Internet of Things: Bridging a Path from CPE 329

Advisor: Dr. John Oliver

By: Steven Han and Rafael Lopez

December 2016

Introduction **3**
Defining the Internet of Things 3
"Prerequisites" 3
Some common IoT protocols 4
Device Choices (for development) 7
IoT Platform Choices 8
Overview of The Big Three's IoT Cloud Services **9**
Continuous Integration of Device and Firmware/Software Management 9
Software Development Kit (SDK) coverage and availability 10
Global Connectivity (2G/3G/LTE…) 11
Support of Standard Protocols 11
Security, Security, Security 11
Large Scale Data Storage, "Cold Storage" 12
Stream/Realtime Data Processing 12
Data Intelligence 13
Conclusion, and our recommendation for Oliver 13
A quick point on swapping cloud providers and hardware (lift & shift) 14
Hello World Example **16**
Lambda: Code in the cloud 16
Serverless framework 17
Creating a new project/application 17
Creating a new Lambda function and deploying to AWS 18
Publishing and subscribing 19
External devices connected to AWS via MQTT 22
Library Tracking Application **23**
Description 23
Project Breakdown 23
Raw Data Collection 24
Data Transfer to Cloud 25
Organization of Data 26
Request Handling 27
Data Analysis 27
Message to End-User 28
Adding New Features to Existing Application 29
Summary 30
APPENDICES **31**

Introduction

“The ability to connect, communicate with, and remotely manage an incalculable number of networked, automated devices via the Internet is becoming pervasive, from the commercial kitchen to the residential basement room to the arm of the fitness buff.” - WSO_2

In this report, we will investigate procedures and technologies used in IoT. A variety of cloud platforms will be described to demonstrate its strengths and usage on IoT applications. Furthermore, demonstrate the most popular hardware being used in several of these applications. This report is aimed to give a good understanding on what it takes to put together an IoT application from from choosing the appropriate hardware to choosing the right cloud platform. A thorough analysis has been done in order to help users choose the *right* hardware, communication protocol and cloud platform to deploy an application to the cloud.

As Internet of Things is still a developing field, the information here may become outdated or even drastically altered, and new standardizations may arise.

This report was completed in December 2016.

Defining the Internet of Things

The Internet of Things (IoT) is the internetworking of smart devices, home appliances, vehicles, internet-connected wearables, and other items with embedded electronics to enable these objects to collect and exchange meaningful data. This data can then be used in big data analysis and machine learning to provide insights to the users. IoT technology has been increasing exponentially over the years as people seek more automation solutions and collect enormous amounts of data.

It has become easier to build applications without the need to purchase and own an extensive set of resources. There are many frameworks available that individuals can use to create their own IoT applications to satisfy needs that have not yet been resolved by a product on the market.

In order to plan and design an IoT application, we first have to know how they should be structured. The following subtopics will discuss that.

“Prerequisites”

Because an IoT application uses a broad range of technologies, it is recommended that you have some previous exposure to the following topics:

- Embedded Systems and Microcontroller programming
- Computer Networks
- Databases (Relational/Non-relational)

- Machine Learning and Big Data Analysis (optional, depends on your application)

A typical IoT system design

The many different types of devices and natures of an IoT application makes it hard to define a *best* way to structure your application. However, ***figure 1*** below is a good general high level design for IoT applications. They comprise of devices with sensors, central servers, and communication pipelines between the components. We will discuss each part in detail later on.

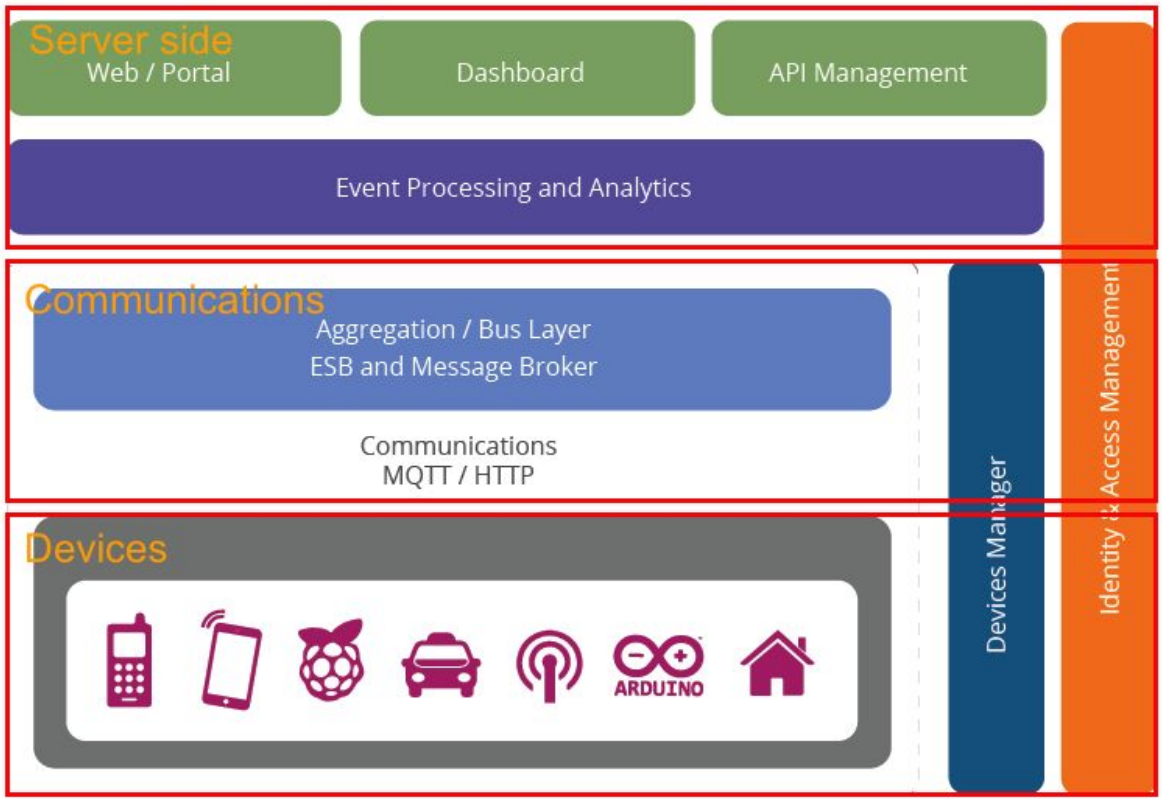


Figure 1. High level design for IoT applications containing devices, communication technologies, and server components.
Source: http://wso2.com/whitepapers/a-reference-architecture-for-the-internet-of-things/

Some common IoT protocols

Since IoT applications can vary a lot in their purpose and usage, it also makes sense that there exists many different communication protocols suited for IoT development. In order to pick when to use which ones, we first have to understand the strengths and weaknesses of each. (Much like data structures in programming)

There are **two** main ways of doing communications:

The ***Request/Reply (or Response) model*** is a basic message exchanging pattern where a requester sends a message to another system, who receives and processes the request, and then returning a response to the original requestor. It allows two parties to communicate with each other. Think of it as a telephone call where one person calls, and the other person picks up. While it is possible to use only request/response protocols in an IoT application, managing a list of *receivers* can quickly become gnarly, and at times, unfeasible.

This brings up the other communication model commonly seen in IoT applicationss - the ***Publish/Subscribe model*.** The publish/subscribe model works like a bulletin board. The publishers do not actually know the receivers of their messages, but rather, the publishers will *post* to a topic, and subscribers of that topic will then receive the messages of interest, without knowledge of the publishers. This pattern allows for better scalability and allows the network topology to be more dynamic.

The following table is a quick summary of some protocols used in the IoT industry. We discuss each of them (except HTTP because you should know it or have a good idea on it) in more detail below, and provide an use case for a more tangible example.

Table 1. Comparison matrix demonstrating features of the mostly used IoT protocols in industry.

Protocol	Transport	Messaging Model	Powered/ Good Connection	Low Power/Lossy Connection	Compute Resources Required	Security
CoAP	UDP	Req/Res	Excellent	Excellent	+	Medium
HTTP	TCP	Req/Res	Excellent	Fair	+	Low
DDS	UDP	Req/Res Pub/Sub	Fair	Poor	+++	High
MQTT	TCP	Req/Res Pub/Sub	Excellent	Good	+	Medium
XMPP	TCP	Req/Res Pub/Sub	Excellent	Fair	+	High (required)
AMQP	TCP	Req/Res Pub/Sub	Excellent	Good	++	High

Source: http://embedded-computing.com/articles/internet-things-requirements-protocols/

Partial sources for content below:
http://www.cs.wustl.edu/~jain/cse570-15/ftp/iot_prot/index.html

Device to Server:

MQTT/SMQTT - (Secure) Message Queue Telemetry Transport

Designed for aggregating data from many devices to be sent to a central server. It is mainly used for send collected data from nodes to the IoT cloud. It is built on TCP for a simple and reliable stream. The secure version has some lightweight encryption, and requires a pre-exchange of public keys (key generation and encryption methods are not standardized yet).

Use case: Thousands of sensors on a underground water pipe sending data to a monitoring server.

XMPP - Extensible Messaging and Presence Protocol

This protocol is more singly addressable than MQTT, using "name@domain.com" addressing schemes. It is not designed to be fast or used for mass messaging, and usually only checks updates on demand. It also runs on TCP/HTTP.

Use case: A home thermostat sending data to a web server, allowing access and control over a phone.

Device to Device:

DDS - Data Distribution Service

With DDS it is still possible to connect devices to servers, but the main purpose is to efficiently distribute millions of messages per second to many simultaneous receivers. Does not use TCP, but instead uses detailed Quality-of-Service control (QoS), multicast, configurable reliability, and widespread redundancy. Devices pool together to implement a "bus" communication, handling data access and updates by many simultaneous users.

Use case: Wind sensors on turbines in a wind farm, processing in real time the wind direction and optimizing which direction all the turbines should rotate to in order to maximize power generation.

Server to Server:

AMQP - Advanced Message Queuing Protocol

Queues and sends very reliable transactional messages between servers. Born from the banking industry, AMQP is focused on not losing messages. Uses TCP as its backbone, and additionally, receivers must acknowledge acceptance of each message.

Use case: Online banking transaction with a single server (HTTP), then that server updates all other servers around the world about that transaction (AMQP).

Device to Server/Device:

CoAP - Constrained Application Protocol ("IoT's HTTP")

A lightweight RESTful interface, which can directly replace HTTP. It is designed to reduce power consumption and transmission overheads in IoT devices. HTTP uses TCP, while CoAP uses UDP with options for reliable or unreliable transmissions. (TCP maintains a connection so it's more power hungry)

Use case: Whenever power consumption should be scrutinized, e.g. battery powered IoT sensors.

Device Choices (for development)

Current boards on the market can be separated into three main classes. A few examples of each are given below:

Basic: These boards usually have system-on-chip controllers. They typically have no operating system on them.

- Arduino Uno: https://www.arduino.cc/en/Main/ArduinoBoardUno
- Arduino MKR1000: https://www.arduino.cc/en/Main/ArduinoMKR1000
- MSP430: http://www.ti.com/lsds/ti/tools-software/launchpads/launchpads.page

Intermediate: Atheros and ARM chip systems fall into this class. They have a limited 32-bit architecture, and sometimes run a stripped down OS or some type of RTOS.

- Samsung ARTIK series: https://www.artik.io/
- TI CC3200: http://www.ti.com/product/CC3200
- Intel Edison: https://software.intel.com/en-us/iot/hardware/edison
- Seeeduino Cloud: http://wiki.seeed.cc/Seeeduino_Cloud/

Advanced: These are the most capable IoT platforms, and have full 32/64-bit architectures. Boards that fall into run a full OS such as Linux or Android (mobile phones can also be included). They can act as a gateway for smaller/basic devices like Fitbits, which connect via Bluetooth to the boards, and then bridged onto the Internet.

- Raspberry Pi: https://www.raspberrypi.org/
- Beaglebone Green Wireless: https://beagleboard.org/green-wireless
- Nvidia Jetson TX1: http://www.nvidia.com/object/jetson-tx1-module.html
- Dragonboard 410c: https://developer.qualcomm.com/hardware/dragonboard-410c

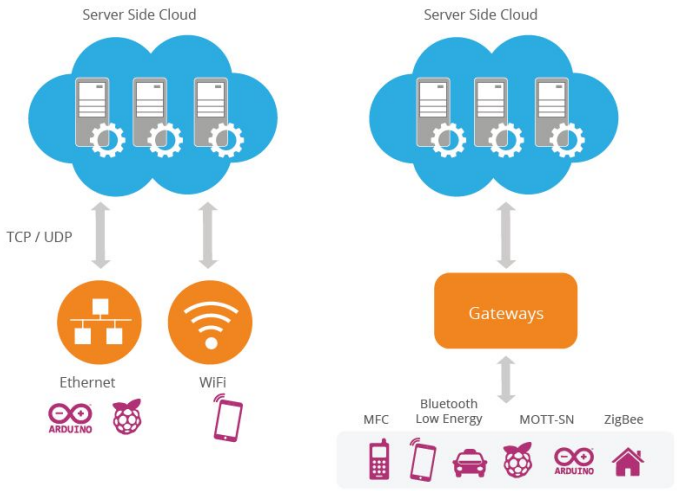


Figure 2. Different techniques devices can use to communicate to the cloud.

Source: http://wso2.com/whitepapers/a-reference-architecture-for-the-internet-of-things/

There isn't exactly a *perfect* board for all intents and purposes, as each manufacturer has their own "ideal" set of features. Fortunately, majority of the boards produced for IoT development have a way to extend their feature set by adding components through pin-outs (though it may be less cost-ineffective).

A more important detail to consider is the languages supported by the board. This is important because it may limit your choice of IoT cloud platforms, which will be discussed next. It is always a good idea to check if your desired platform has an application programming interface (API) in the language you need for your board.

IoT Platform Choices

It should be evident from the previous sections that an IoT application would require the developer to have a server in order to aggregate data from the sensor devices. Obtaining a physical server and connecting all the backend stuff would be costly, and a lot of work. It is also troublesome to maintain a High Availability (HA) server. Therefore, most developers and even large companies turn to cloud platform providers to do the messy work for them.

For curiosity purposes, these are some things that a cloud platform should provide:

- Distributed computation servers
- Sharding databases
- Data analytic services
- Machine learning services
- Secure communications and transmissions
- a few other services that are more *optional* (e.g. long term storage, web hosting)

And finally, a main *hub* that connects/ties all the above services together.

The choice for a *right* platform is dependent on aspects such as security, flexibility/scalability, data intelligence, and cost (in order of importance). Furthermore, a certain platform might specialize on certain aspects and provide little to no support on another. For example, a platform might provide high security components but limit its use to one communication protocol.

Here are some of the platforms that are currently available for public use:

- **Google Cloud Platform**: https://cloud.google.com/solutions/iot/
- **Microsoft Azure**: https://azure.microsoft.com/en-us/solutions/iot-suite/
- Salesforce Einstein: http://www.salesforce.com/iot-cloud/
- Xively: http://xively.com/
- **Amazon Web Services**: https://aws.amazon.com/iot/
- IBM Watson: http://www.ibm.com/internet-of-things/

Out of these services the top three platforms that are mostly used are Google Cloud Platform, Microsoft Azure, and Amazon Web Services. Correlated to their popularity, community support for these platforms is also more accessible (e.g. stackoverflow.com). AWS has enjoyed a head start (in 2004!) over the other two in terms of running a cloud platform, but these three providers are going to stay as the top three for the foreseeable future simply because of their popularity.

If you decide to venture into the other newer providers, it is almost mandatory for you to read up on each platform to see what functionalities each offers, and understand their capabilities. The big three is pretty similar though, most functionalities just have different names. A (huge) comparison matrix is attached as ***Appendix 9*** at the end of this report.

Overview of The Big Three's IoT Cloud Services

In the previous section we outlined the required capabilities of an IoT platform. Now, let's dive into a more detailed comparison of the top three cloud providers, and how their provisions matchup with each other.

It should also be noted that Google's IoT platform (Brillo) is still in an "invite only" phase.

Continuous Integration of Device and Firmware/Software Management

A good IoT system architecture should allow for developers to continuously update their sensor devices' software over-the-air (OTA). This is quite an important capability - imagine having to physically connect to thousands of sensor boards attached to an underground pipeline every time a bugfix is performed. What a nightmare.

The Big Three have years of experience under their belt with Echo/Kindle, Windows/Xbox, Nexus/Android. There is no clear winner here, as they all have services specifically for this - and you can be assured their services do this too easily for you.

- AWS offers their continuous integration and delivery service with *Code Pipeline*.
- Google has *Brillo* to manage device firmware and updates.
- Azure has the *Windows Update Service* in their Windows IoT core.

Software Development Kit (SDK) coverage and availability

Obviously, an IoT platform is pretty useless if it doesn't support developing with your preferred device. Development Kits, or *devkits*, are what you need for this. If the provider has a devkit with the programming language your device supports, then all is well. If not, then the developer will need to manually write code to integrate the services and tie them to their devices. This translates to a bogged down (and definitely buggy) workflow, and nobody likes trudging through that.

- AWS offers Embedded C, Python, Javascript, Objective-C (iOS), Java, Android, and Arduino Yun. Their SDKs are well adopted by many chipset manufacturers, seen from the number of their recommended microcontrollers.
- Google Brillo has less conventional SDKs for IoT development on microcontrollers, supporting a number of languages like Go, Android, .NET, Javascript, Objective-C (iOS), PHP, Python, and Ruby. Their SDKs are also a little spotty with supporting backend IoT data management and analytics.
- Azure has SDKs for .NET, Javascript, Java, C, and Python. It is less than the other two, but since microcontrollers usually run with C, it still comes out above Google's Brillo. W10 IoT core is also part of the *Universal Windows Platform (UWP)* environment, which makes things easier for current Windows developers.

Given Amazon's head start in the cloud computing platform, they edge out the competitors in this regard. The other two has some catching up to do. Google is definitely leaning more towards mobile phones for IoT, while AWS and Azure covers microcontrollers too. Azure is a natural choice if using W10 IoT devices, or else AWS should be the way to go here. It should also be noted that AWS has a number of open source frameworks (top 3 on Github!), such as *Serverless*, that help you with scaffolding your project. The guide at the end of this write up uses *Serverless* to set up a project with AWS.

Global Connectivity (2G/3G/LTE…)

Sometimes, WiFi can be scarce - I'm sure we've all experienced this. Having the support of telecommunication networks in the IoT stack can be valuable, especially if your IoT application is deployed in a less urban area. A mobile virtual network operator (MVNO) allows the use of any nearby radio towers for your devices' internet connectivity, doesn't matter if it's AT&T, Verizon, or T-Mobile's radio tower. At this moment, it looks like Google is the only one implementing this and leading the way with *Project Fi* and *Project Loon*. Not too serious, but it comes in handy. Hopefully the other two will join in soon.

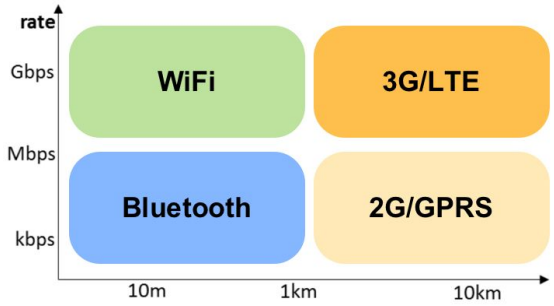


Figure 3. A rate-range chart of the different connectivity technologies.

Support of Standard Protocols

Right now, it looks like there are as many IoT protocols as there are IoT platforms. Which ones should you use? Nobody knows. The IoT development community is pretty much waiting to see which of these deployed protocols dominate the others.

- Google Cloud Platform is pushing *gRPC*, *Weave* and *AllSeen*, which are the less used "standards". They are comparable to MQTT/AMQP/etc, it's just that developers have gotten used to the more mature protocols and are reluctant to switch.
- AWS uses the tried and tested MQTT/AMQP/XMPP protocols, which seems to be the majority so far.
- Azure's Windows IoT has an open source MQTT client that developers can use.

We think AWS wins this category, just because they've enjoyed a monopoly before cloud computing became mainstream, and have a larger number of users. Their choice of protocols have sort of become the de facto standard for this industry.

Security, Security, Security

Security is of the utmost importance in all IoT designs (I can't stress this enough). All layers of the IoT stack should have exceptional support for security, no exceptions. Malicious hackers can do serious damage to just about anything, from collecting data about your home (your in/out

timeframes), deregulating gas pipe flow rates, to creating a nuclear meltdown at a generation plant.

- Google Brillo has strong, default required security at the hardware and software levels. Baking in security from the start is always better than implementing them afterwards.
- Azure has been making large security improvements in their IoT suite, and will continue to do so according to their blog posts.
- Amazon's IoT Hub and their frontend/backend interconnections offer a strong security model, using authentication/certificates/authorizations.

Hard to go wrong with any of them here. They all agree that security is paramount to the IoT design.

Large Scale Data Storage, "Cold Storage"

Often, during large scale IoT application usages the developer may find themselves knee deep in data, which might or might not be useful later. An IoT platform should allow storing data in large amounts, in the magnitude of gigabytes or terabytes instead of forcing the developer to prune their datasets. They should also allow storage in different latency tiers (slower = cheaper), since data archives might not be accessed for a long time, if ever.

- Azure has *DocumentDB, StorSimple, Data Lake Store, Redis Cache*, just to name a few. They have all kinds of storage needs you might need, and an Event Hub to help sort your data.
- AWS has *DynamoDB, Redshift, Glacier, Aurora, S3* and more that can handle scalable data management and archiving requirements of small to large IoT applications.
- Google Cloud also has a number of choices such as *Cloud Storage, Cloud SQL, Cloud BigTable, Cloud Datastore, and Persistent Disk*.

All three providers are on par in this regard. They have all the storage needs you will ever need.

Stream/Realtime Data Processing

A typical IoT application generates a flood of data to the cloud platforms. It would be ideal if the cloud platform supports stream data processing and makes it part of the IoT data intelligence design. Streaming data needs to be processed sequentially on a record-by-record basis, or over a sliding window, which are then used for analytics like aggregations, correlation and sampling. If not, data will be actually stored first, and then pulled out at a later time to be analyzed. This stream processing saves time, and even reduce latencies of your application.

- Amazon Kinesis is AWS's stream data processing service.
- Google uses a combination of Cloud Pub/Sub and Cloud Dataflow to do the same.
- Azure has Stream Analytics to process data streams.

Again, can't go wrong with either of them. They all provide this service, and are also probably the best at what they do given the scales their own companies operate at.

Data Intelligence

A successful application will require the developer to make sense of the massive amounts of data collected from the sensors. Data is useless if you cannot find meaning in it, and doing so is probably the pinnacle of IoT and what IoT tries to provide/solve.

- AWS offers Elastic Compute Cloud (EC2), Lambda, Elastic MapReduce(EMR), Kinesis Analytics, Rekognition, Athena, and a few more. Their services tie together pretty well with their IoT hub, which makes it convenient for developers.
- Azure too, has a vast number of services like HDInsight, DataFactory, Data Lake Analytics, Machine Learning… just to name a few. Their integration is not as lauded by the IoT community as AWS's, but still quickly usable nonetheless.
- Google's situation with their Data Intelligence services has left the IoT community with mixed feelings. On one hand, their analytics are superior to the other two, with Cloud Dataproc, Cloud Datalab, Cloud Machine Learning, Prediction API, Google Genomics, Compute Engine, Google DeepMind (neural networks AI!) in their toolbox. However, there is almost no integration of their superior data intelligence services to their IoT platform. Developers will have a hard time trying to use these more sophisticated intelligence services.

The competition is tight here, but we are probably giving this win to AWS too, according to public opinion. We could not quite test this part, simply because of the level of difficulty and the lack of real world data available from the libTracker project below.

Conclusion, and our recommendation for Dr. Oliver

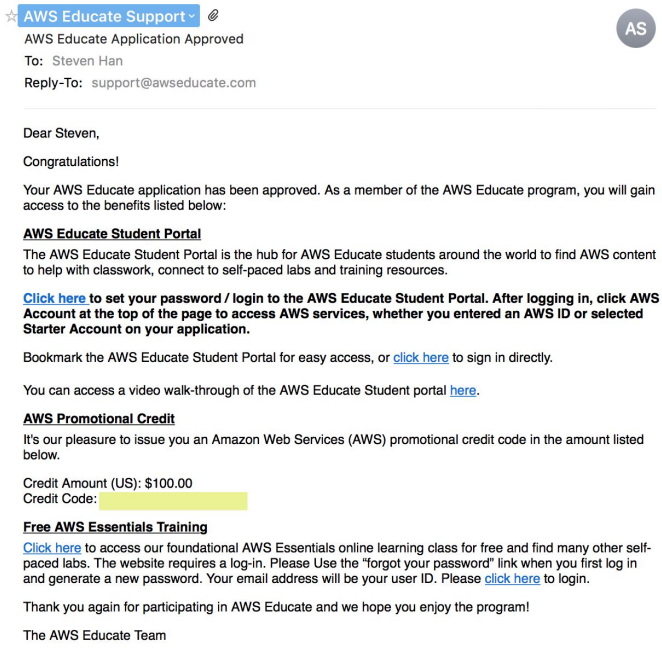
While Google Cloud Platform had good tools required for an IoT platform, it felt more like they were made for experienced developers who knew what they were doing, what they wanted to do, and what they needed to do. Their cloud services were not very integrated, and developers have to get their hands dirty in order to connect services together. It might be a good thing for developers who don't like restrictions about how they structure their application, but it will definitely drown a developer who is just starting out in IoT with the amount of backend work required to say "Hello World" (if they even figure out what they have to do).

Azure and AWS are almost on par with each other, with Azure only falling a little bit behind AWS in most aspects. We feel that a beginner could use either one of them and not end up in tears like if they chose Google Cloud. The only reason we would recommend Azure is if the developer intends to work with Windows 10 IoT Core devices, because other than that, AWS can match them stride for stride, and usually even a little better (with more support resources available).

As much as we would have liked to have tested all three platforms to give you the best information, the truth is that AWS seemed the most appealing to us from the start. It was by all means not easy to figure out what things we would need to do to get started with AWS, but it was the easiest out of the three to find information on. Azure was our next choice, and Google got thrown out pretty quickly because we had no idea how to start to connect their services together. We did not even know the best practices for developing an IoT app, let alone designing it from scratch using Google.

We also have a couple of (useful) things for you that we found while messing around with AWS:

1. AWS has an AWS Educate portal! https://aws.amazon.com/education/awseducate/
    - We found this a little too late… But the material inside looks much more organized and useful compared to the normal tutorials offered by Amazon.
    - They also give you $100 credit, in addition to the free tier stuff that's already available. Just in case you want to test out/learn some services that are paid-only. (e.g. Machine Learning)
    - AWS also has a group management function, which we think you'd be able to set up a Teacher(admin) - Student(user) sort of structure, and share an AWS account for the whole class. Not quite sure of its capabilities and possible consequences, but we thought you could take a look.



AWS Educate Support

AWS Educate Application Approved

To: Steven Han

Reply-To: support@awseducate.com

AS

Dear Steven,

Congratulations!

Your AWS Educate application has been approved. As a member of the AWS Educate program, you will gain access to the benefits listed below:

AWS Educate Student Portal

The AWS Educate Student Portal is the hub for AWS Educate students around the world to find AWS content to help with classwork, connect to self-paced labs and training resources.

Click here to set your password / login to the AWS Educate Student Portal. After logging in, click AWS Account at the top of the page to access AWS services, whether you entered an AWS ID or selected Starter Account on your application.

Bookmark the AWS Educate Student Portal for easy access, or click here to sign in directly.

You can access a video walk-through of the AWS Educate Student portal here.

AWS Promotional Credit

It's our pleasure to issue you an Amazon Web Services (AWS) promotional credit code in the amount listed below.

Credit Amount (US): $100.00
Credit Code:

Free AWS Essentials Training

Click here to access our foundational AWS Essentials online learning class for free and find many other self-paced labs. The website requires a log-in. Please Use the "forgot your password" link when you first log in and generate a new password. Your email address will be your user ID. Please click here to login.

Thank you again for participating in AWS Educate and we hope you enjoy the program!

The AWS Educate Team

A quick point on swapping cloud providers and hardware (lift & shift)

If you take a look at **Appendix 9**, it should be easy to see which services are equivalent to each other across the different cloud providers. It would then come down to moving the code written for one provider to the other, which would include swapping out the SDK libraries and refactoring code to fit the new SDK (the API calls). It should be fairly straightforward going between AWS and Azure, as their structures are similar. However, we cannot say the same for Google Cloud because they have some differences in structuring the architecture of your application (GCP is more mobile oriented).

Hardware wise, as long as your new board supports the language your code is in, then it should only come down to remapping the pinouts from the old board to the new board. (eg. GPIO23 on Raspberry Pi board A is GPIO10 on Raspberry Pi board B)

Hello World Example

IoT and cloud platforms sound very abstract from a high level since the components appear to be very complex. A small example has been put together to demonstrate a simple IoT application, to depict some of the computational capabilities of a cloud platform connected to an external device. For this example, Amazon Web Services (AWS) was chosen as the cloud platform of choice and a Raspberry Pi as the hardware of choice. A Raspberry Pi will take input from a button and deliver "Hello World!" to an email using AWS. This will be built step by step starting from cloud configuration to interfacing the Raspberry Pi with AWS.

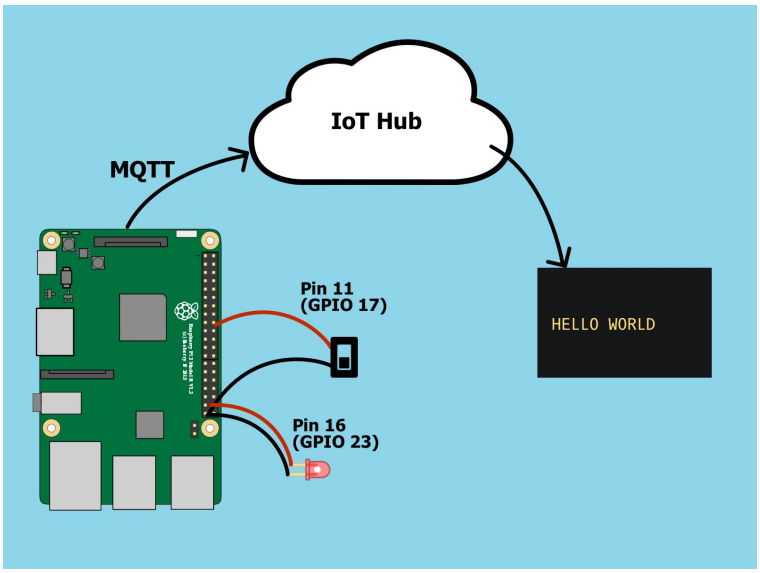


Figure 4. Hardware setup for the Hello World application displaying GPIO pinouts.

Lambda: Code in the cloud

Lambda is a service by AWS in which users can run code in the cloud without providing or managing servers. Lambda functions can be made up of single or multiple functions. Its main attempt is to execute code that is specific to a certain event. For example, when event A happens, trigger lambda A1, and when event B happens, trigger lambda A2. An event can be any particular scenario that is meaningful enough to invoke an action i.e. a HTTP request. This is just a glimpse of how distributive computing works in the cloud.

Making a lambda function can be accomplished through the AWS Console. https://aws.amazon.com/lambda/. From the console, a user can select 'Create a Lambda function' and follow the on-screen steps to create and upload code to be run in the cloud. This method of creating a lambda function is easy to use, but as a particular project becomes larger, it becomes tedious and harder to manage. A better approach is to use Serverless architecture built to work alongside AWS. https://serverless.com/

Serverless framework

The Serverless architecture makes it really simple to manage AWS applications using Lambda functions. Serverless facilitates the development, testing, and deployment of the lambda functions within an AWS project. Users don't have to worry about specific Lambda configuration, as Serverless handles it in the background. The lambda development can occur directly from the command line with little usage of the AWS console. Serverless is one of the most popular open source projects on GitHub where developers can easily create and incorporate plugins to use in their applications. Furthermore, it is a framework that is widely used in industry by companies such as Black & Decker, Verizon and Amazon of course.

Serverless is a ***Node.js*** command line interface tool, so it requires Node.js to work.
To install Node.js: https://nodejs.org
To install Serverless:
https://github.com/serverless/serverless/blob/master/docs/providers/aws/guide/installation.md
**Note: Version 0.5.6 is recommended to avoid any problems.

The documentation for version 0.5.6 can be found at https://github.com/serverless/serverless/tree/master/docs . It provides specifics on the files created upon a new project or lambda creation. Highly recommended to be read.

Creating a new project/application

Once Serverless was successfully installed, a new project can be created. However, this project is unknown to AWS, it just serves as a way to organize functions better.

To create a project start by typing in the command line:
----> *serverless project create*

The command line will prompt for:

1) Project Name
2) The stage name (usually 'dev')
3) The AWS Profile to use (having already registered to AWS online)
4) Region (what data center to use to run code on)

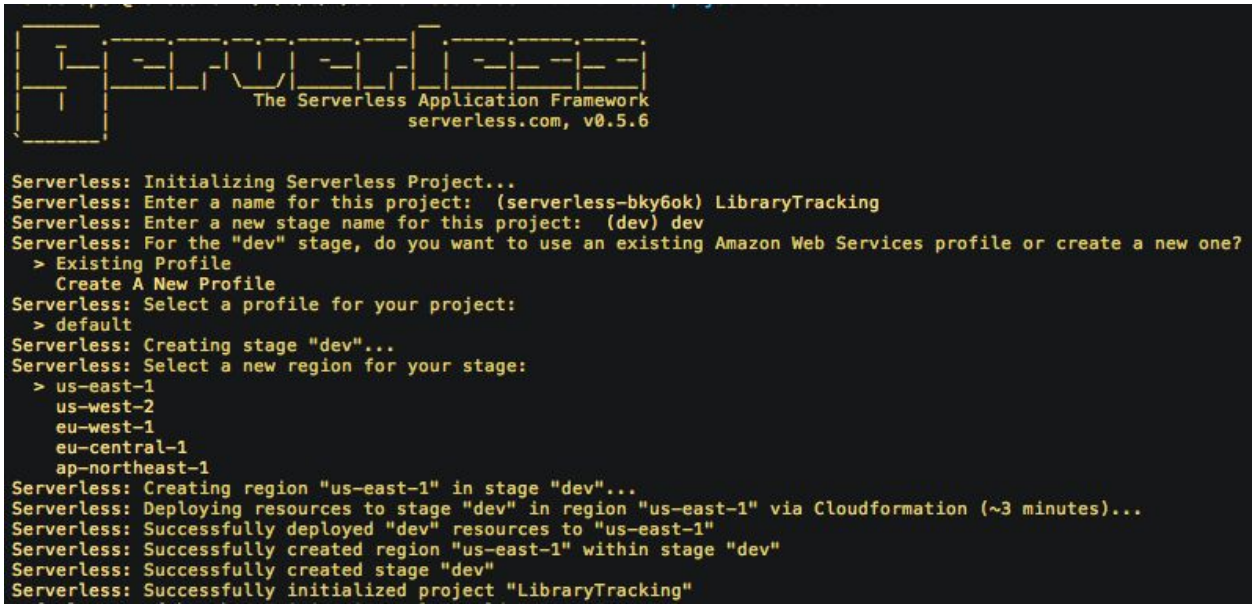


Figure 5. Quick look at how serverless creates a project for AWS with given server constraints (i.e. region, and developing stage)

This will have created a folder for the desired project. Under the same folder, it is recommended to make a new directory named 'functions', to store any Lambda functions that can exist in the project for good organization.

Creating a new Lambda function and deploying to AWS

To create a lambda function:
----> *serverless function create*

This will prompt user for:

1) Function Name
2) The language of the function
3) Configuration of Endpoints or Events***

****Note***: Function can be created without specifying endpoints and events. It can be configured at a later time, as project is developed.

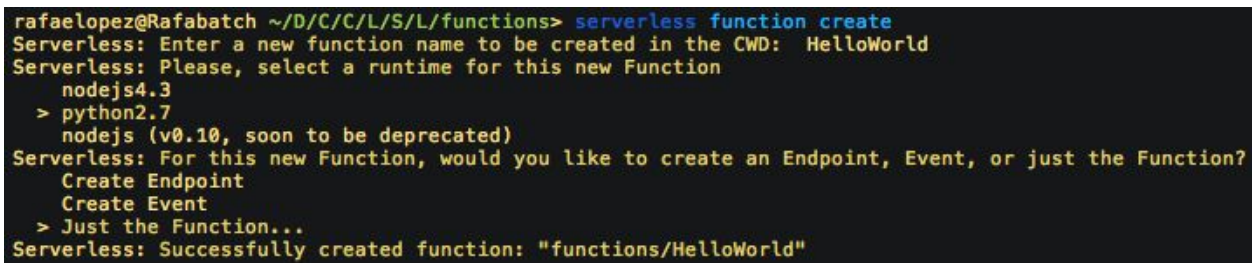


Figure 6. Serverless attempt to speed up Lambda creation as described above.

Python2.7 has been chosen as the language for the lambda function. Serverless created a folder containing ***handler.py.*** This will contain the code of the lambda function created. The next step will be to edit this file to make the function return "Hello World!".

Testing this new change can be done by doing:
----> *serverless function run*

```
rafaelopez@Rafabatch ~/D/C/C/L/S/L/f/HelloWorld> serverless function run
Serverless: Running HelloWorld...
Serverless: ------------------
Serverless: Success! - This Response Was Returned:
Serverless: "Hello World!"
```

Figure 7. Serverless method of testing a Lambda function locally.

This will mimic the behavior that AWS Lambda does when it attempts to execute the provided code. When the local testing is complete, it can be deployed to the cloud. To deploy the newly created lambda just do:
----> *serverless function deploy*

```
rafaelopez@Rafabatch ~/D/C/C/L/S/L/f/HelloWorld> serverless function deploy
Serverless: Deploying the specified functions in "dev" to the following regions: us-east-1

Serverless: ------------------------
Serverless: Successfully deployed the following functions in "dev" to the following region
s:
Serverless: us-east-1 ------------------------
Serverless:   HelloWorld (LibraryTracking-HelloWorld): arn:aws:lambda:us-east-1:8362726571
74:function:LibraryTracking-HelloWorld:dev
```

Figure 8. Serverless deploying function to AWS as a Lambda function.

This will actually 'create' the lambda function on AWS with the name: 'ProjectName-FunctionName' making it easier to denote which lambda functions belong to certain projects. By accessing https://aws.amazon.com/lambda/ , the newly created function will be listed in the functions. The lambda function can also be tested on the console and it should return the same result obtained when testing locally. This is the basic procedure to get a lambda up and running on AWS.

Publishing and subscribing

In IoT, there is a big emphasis on the idea of topics. Topics consist of publishers and subscribers. A certain subscriber sends a message (publishes) to a topic, and any devices listening, or rather subscribed, to that topic will receive the message sent by the publisher. This is the main concept of the MQTT protocol, in which a publisher sends a message to a broker, which handles the incoming messages, and delivers them to the appropriate recipients Below is a good representation of this idea.

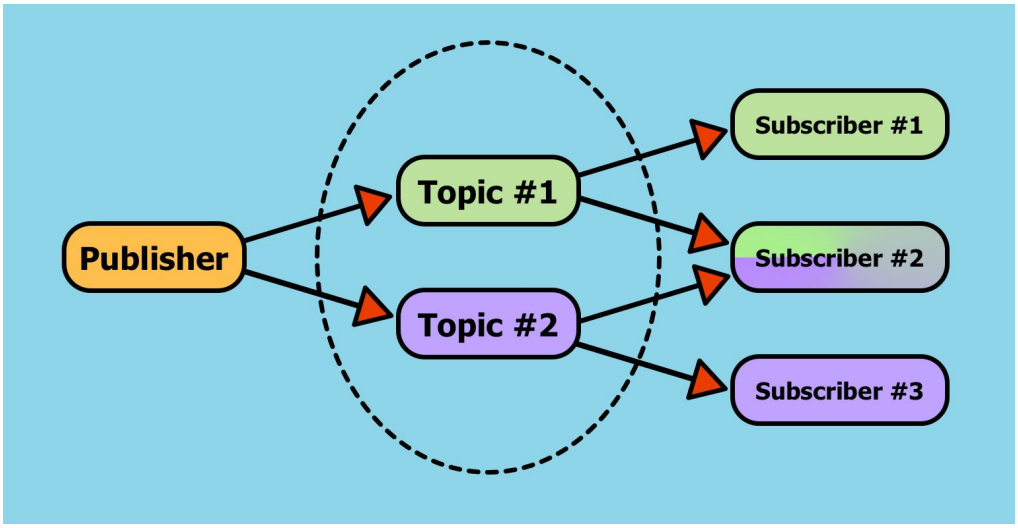


Figure 9. An example of a topic. A publisher sends a message to a topic, that gets received by publishers. A subscriber can be subscribed to 1 or more topics.

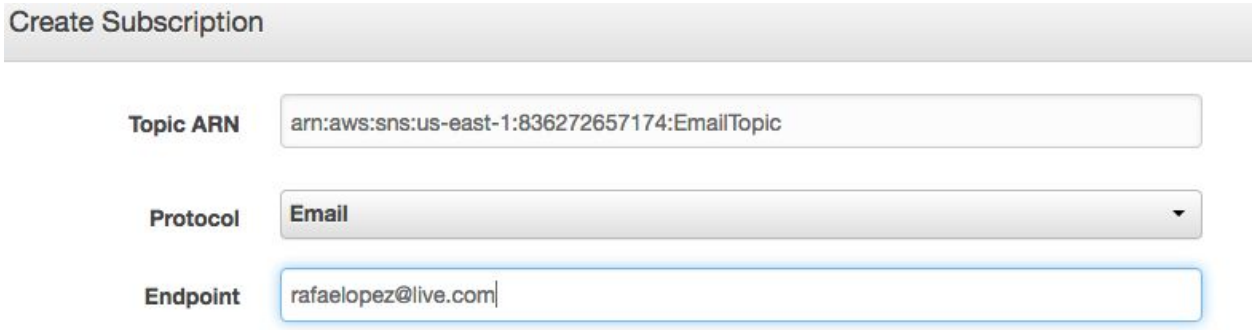


Figure 10. Creating a SNS topic in AWS. Devices subscribed receive whatever is published to this topic.

The current Hello World project will be modified to include SNS and send "Hello World" to an email, rather than returning it to screen. To add subscribers to this newly created topic, the option of 'Subscriptions' must be accessed on the SNS console. In there, several message protocols can be configured. Some of the protocols are: HTTP, Email, SMS, AWS lambda (function to be executed in the cloud). For this setup, the Email protocol will be used, and an email address will be served as the endpoint on the topic. It might be required to confirm subscription to this topic via the email desired as endpoint.

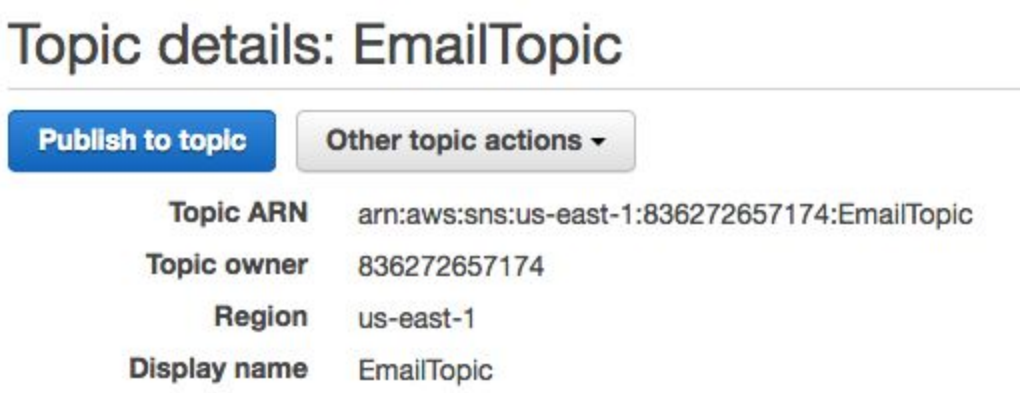


Figure 11. EmailTopic created. Any client attempting to publish to this topic, will have to do so through the required Topic ARN.

After having set up this configuration, the Lambda function created, handler.py, can be edited to publish a message to the topic. To accomplish this ***boto3*** will be used. Boto3 is the AWS SDK for Python. This SDK provides the capability of using several AWS services via code. Specific documentation on *boto3* can be found at: https://boto3.readthedocs.io/en/latest/ .

The code to be added to handler.py is the following:

```
import boto3

client = boto3.client('sns')
outputMsg = "Hello World!"

publishResponse = client.publish(
    TargetArn = 'arn:aws:sns:<region>:<AWS Acct Number>:EmailTopic',
    Message = outputMsg
)

return publishResponse
```

***Note**: TargetArn is the ARN obtained after having made the EmailTopic topic. It is recommended to copy and paste directly from the console to avoid any discrepancies.

After making the correct code changes and re-deploying to AWS, the message: 'Hello World!' can delivered to the desired email. Test this by using the 'Test' option under the Lambda console on AWS.

External devices connected to AWS via MQTT

At this point, several IoT concepts have been used to do work on the cloud but it still isn't a true IoT application until external devices are added. To keep this current application simple enough, a single external device will be included. This will be Raspberry Pi Zero connected to a button. At the press of a button, the original "Hello World" message will be delivered.

The next steps assumes that a Raspberry Pi has been already setup and can be controlled via terminal. The library wiringPi has been used to control the hardware and to make the code easier to write and organize. The following links provide information on how to install the library and how to use it:
https://projects.drogon.net/raspberry-pi/wiringpi/download-and-install/
https://learn.sparkfun.com/tutorials/raspberry-gpio/c-wiringpi-api

Before connecting the Pi to AWS, make sure that the hardware and software is properly set-up to react to a button press. To test the software functionality, an LED will be turned on everytime a button is pressed. Source code for this can be found on ***Appendix 1***.

The expected behavior is similar to the previous LED example. Instead of blinking the LED, a message will be sent to AWS to notify about a button press via email. Before this, AWS will have to know about which 'things' are to be connected to the cloud. This involves using the AWS IoT service. On this service, a user can create a resource that will represent the 'thing' on the cloud. In this case the Raspberry Pi. The reason to do this, is for AWS to provide appropriate certificates for secure communication. A 'thing' that is not registered, does not receive certificates and is not granted access to communicate in the cloud. AWS will grant certificates depending on the SDK used for development. This application will use the embedded C SDK. This SDK provides code to communicate to AWS. Connecting the Raspberry Pi to AWS will be handled by the MQTT protocol. OpenSSL and OpenTLS are open source libraries thatcan be used as the communication channels to enforce this protocol. AWS provides code to use OpenTLS in an application very easily. The code and examples can be found at:
http://docs.aws.amazon.com/iot/latest/developerguide/iot-device-sdk-c.html

The SDK already includes OpenTLS internally, and it just a matter of copying the certificates obtained from AWS IoT in the proper folder for the SDK to reference to. The folder is '***/certs***'. Amazon provides source code to demonstrate sample applications using the embedded C SDK. Reading through the libraries in the folder is recommended for specifics. ***Appendix 8*** contains source code that integrates the Raspberry Pi + button code, with OpenTLS. The major point in this code, is to understand that the Raspberry Pi will act as a *publisher* in the MQTT communication to AWS. The Raspberry Pi will be publishing to a topic, i.e. ***mqttTopic***. Any subscribers to this topic, will receive the message of "Hello World" as well as the number of times that a button has pressed.

To include this change in the on-going project, a new SNS topic must be created. The source code assumes and publishes to ***mqttTopic***. After having created this SNS topic, a new subscriber needs to be added. Just like how the Email subscriber was configured, this subscriber will be of type *Lambda*. It just means the lambda function wants to be triggered and use the message published to the topic. This lambda function is the previously deployed function to AWS. This function must edited with minimal changes to use the newly obtained message. On ***handler.py***, make the following change before the ***boto3*** client publishes:

```
outputMsg = event['Records'][0]['Sns']['Message']
```

This will access the payload portion of the SNS section in the event that triggered the lambda. Remember, a lambda is triggered by an event, and that event can vary, and contain several other fields. Within the 'event', a 'Records' portion exist that contains information about the type of the event and its specifics. It's always good to print the whole event object at first to familiarize with the format. For specifics on this, it is recommended to read on AWS Lambda invocation via SNS: https://aws.amazon.com/blogs/mobile/invoking-aws-lambda-functions-via-amazon-sns/

Make the changes, save, and deploy to AWS. At this point, every time a button press is recognized by the Raspberry Pi, an email will be received saying "Hello World!" as well as the number of button presses.

Library Tracking Application

Description

Cal Poly's Kennedy Library experiences heavy loads of student traffic throughout its hours of operation. At certain times, Cal Poly students will decide to chose the library as the location to study. However, it is often very difficult to determine the current state of the library as it varies throughout the day. Sometimes students spend 5-10 minutes walking around the library looking for a location to study. Sometimes they are lucky, sometimes they are not.

There should be a solution to this. A student should be able to request information on the state of the library at certain times of the day to determine if the Kennedy Library is the best option to study at a specific time of day. The next section attempts to resolve this issue using a more centralized IoT approach.

Project Breakdown

The end users care about the current state of the library, that is, an estimate of how full the library is, as well as a recent estimate of the student traffic. Student traffic, focuses on how many students have entered or exited the library per minute. To obtain this information, a system must

be developed that accurately calculates how many students exit or enter the Library. Such information should then be delivered to the user.

A solution for this problem statement relies on the following phases:

1. Raw Data Collection
   - Detect when students enter or exit the library.
2. Data Transfer
   - Send collected data to the cloud.
3. Organization of Data
   - Store the received data for future analysis.
4. Request Handling
   - Detect and take action when a user wants to know information about the state of the library
5. Data Analysis
   - Determine out the current state of the library
6. Message to End-User
   - Let the user know about the current state of the library in a user-friendly message.

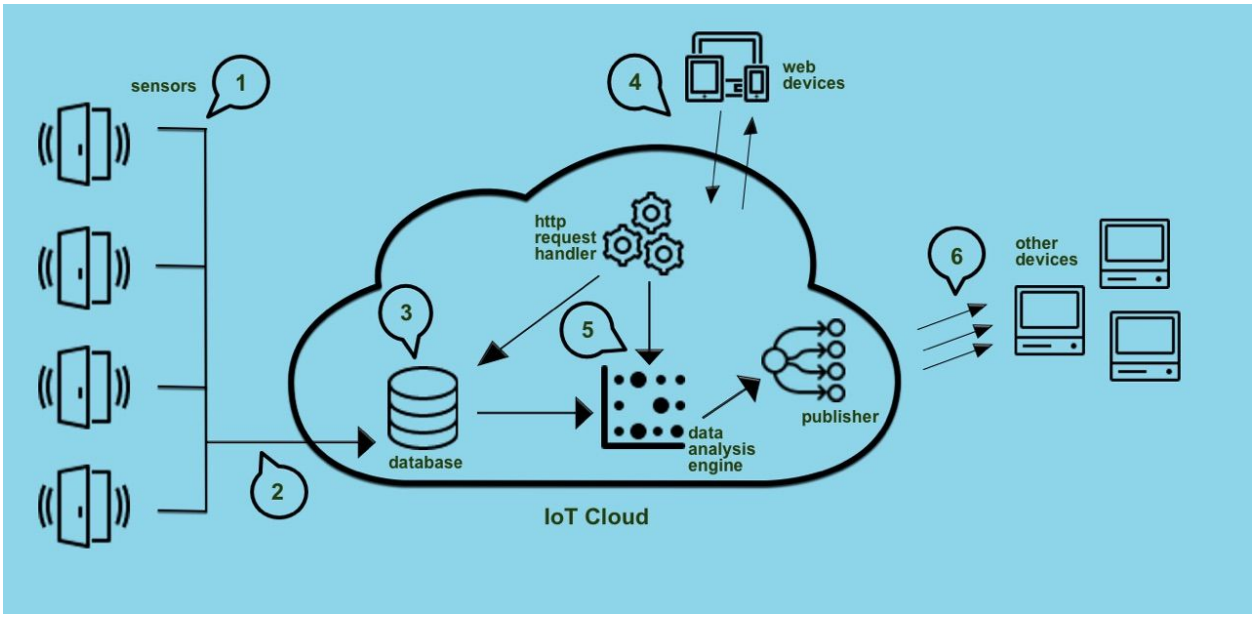


Figure 12. High level design of the application. Phases demonstrating the expected behavior of the system. Both the microcontroller and the user, can perform separate actions on the cloud.

Raw Data Collection

Data will be collected at the two main entrances of the Kennedy Library. Student movement can be either 'entering' or 'exiting'. The system should properly keep track on how many students have been entering/exiting on certain time intervals and send that information over to the cloud.

The microcontroller chosen to handle the data detection was the Raspberry Pi because of its low cost and portability.

There are various technologies that can make the traffic detection very precise. For example, using a camera along with ComputerVision. Using this approach can be costly and be heavy on the processing side because of the restrictions on the Raspberry Pi. To remain within the scope of this project, a different approach was developed to handle the student traffic.

The alternative of this was to develop a data collection algorithm that would be of low-cost and low in processing and that was accomplished with the use of motion sensors. Because of the aims of this algorithm, the data collection is prone to errors. Refer to ***Appendix 7*** and ***Appendix 6*** for the source code and software diagram depicting the behavior of the algorithm.

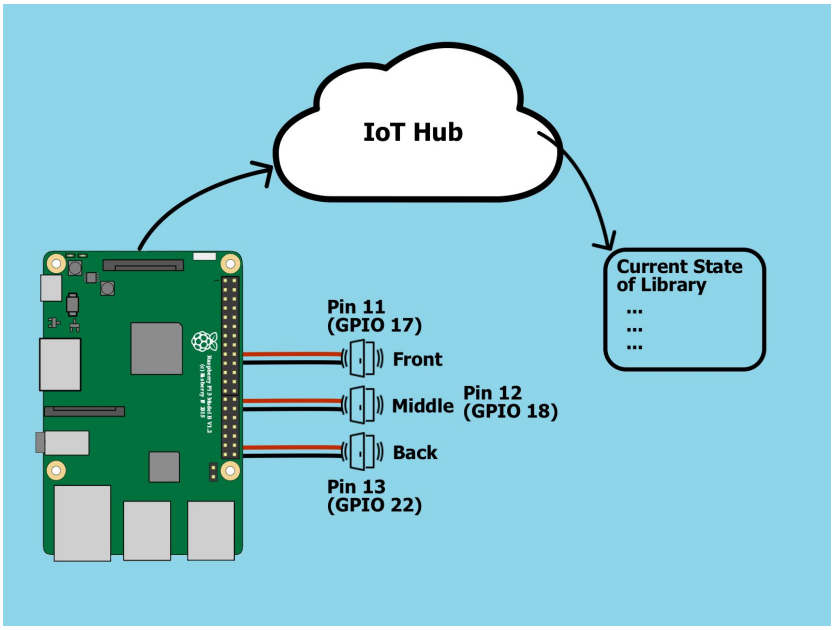


Figure 13. Hardware setup for the library application. Three motion sensors linked to Raspberry Pi via GPIO ports.

The major concept to grasp from this section is to understand that the Raspberry is capable of detect when a student is entering or exiting upon the trigger of motion sensors. All this is handled on the hardware side, and no cloud communication is used yet.

Data Transfer to Cloud

After data has been collected by the microcontroller (people entered, people exited, current number of people inside) it will be transferred over to the cloud to be recorded somewhere. The data transfer follows the same format used in the 'Hello World' application (***Appendix 8*** for reference). The Raspberry pi will act as a publisher publishing to a topic handled by SNS. A new SNS topic will have to be created: ***rawDataTopic***.

The Raspberry Pi will publish a JSON object to the topic containing the following fields:

1. Timestamp of record
2. Location of record
3. Number of people that have entered the library
4. Number of people that have exited the library

The location field will be initialized depending on which Raspberry Pi is sending the information. Multiple Raspberry Pi's will be sending information depending on which entrance are monitoring. For example, one will be sending 'MAIN' as its location, and another '24HR' describing data obtained from the 24 hour entrance. The code loaded into each Raspberry Pi would be the same, just differing in the location field that each is ending. This implementation thus makes this application very scalable and easier to implement.

This JSON object will be the payload message to deliver. Here is an example on how this JSON object would be built within a buffer.

sprintf(buffer, "{timestamp: %d, location: %s, in: %d, out: %d}", unixTS, loc, peopleIn, peopleOut);

The code in ***Appendix 7*** will have to be merged into ***Appendix 8*** to sections 1 and 2 described by ***figure 12***.

Whatever client is subscribed to this *rawDataTopic* should be expecting a message to be in this format. The message will be handled by a Lambda function on the cloud subscribed to this topic. It will use this message to store into a database.

Organization of Data

After the Raspberry Pi has successfully published the JSON object to the SNS topic, the message can be parsed and stored into the cloud. The storage will take place in a NoSQL database for performance and scalability. Amazon provides this service via its DynamoDB service. Reference https://aws.amazon.com/dynamodb/ for details on DynamoDB.

Below is a schema of the major table where the data from the Raspberry Pi is stored.

Table 2. Table RAW_DATA. Demonstrates the major collected points at various times and locations.

id	timestamp	location	in	out
336	1479338369	MAIN	35	56
337	1479310102	24HR	66	22

Before inserting, a table should be created on DynamoDB. This can be done through the DynamoDB console on AWS console or via a code. ***Appendix 3*** provides the source code to create a DynamoDB table using boto3. It is important to denote that a sorting key and partition key are only needed for creation, other column fields could be added when inserting new elements into table. Partition key is the id value and timestamp is the sorting key. This lambda function can be executed by itself as its only a one time thing.

A new lambda function will be required to insert a function into the RAW_DATA table. This lambda function will subscribed to the SNS topic *rawDataTopic*. The source code to handle this action can be located on ***Appendix 4***. It is important to make sure that this lambda function is subscribed to *rawDataTopic* or else the code will never get executed leading to no insertions into the table.

The newly created Lambda function will have to obtain permission to access the database because AWS sets restrictions on what users, functions and service can do within AWS. This is to enforce security and make sure certain services don't have access to certain information. To obtain permission, access the ***Identity and Access Management*** service on AWS. Look on *roles* tab, and look for a role that contains a name similar to the Lambda function needing to get access. Serverless has already created a role for the Lambda function. Under *Manage Policies*, select the policy required. (Tip: Search for whatever service name access is needed, a list will appear of policies for that service)

These 3 completed sections will be be what the Raspberry Pi and the cloud be doing as a standalone subsystem: capturing data, sending over data, and inserting data into a database. This is required for when a user requests analysis on the state of the library. The user interaction will be handled in the next section.

Request Handling

Handling the request of a user involves the execution of code on the cloud to produce a proper. This can be if the request is made via a web application or a phone application. AWS accomplishes this via its internal service API Gateway. The information and documentation can be found at https://aws.amazon.com/api-gateway .

API gateways essentially tells AWS to perform a specific logic in the cloud. In this library tracking application, it will trigger a lambda function, labeled by number 5 in ***figure 12***, to start the analysis portion of the request. This in fact initializes the chain reaction of lambda functions that will in fact deliver the message to the user. API gateway is the start of the second subsystem of the application of which is unaware of the Raspberry Pi.

Data Analysis

This portion of the application will retrieve information from the database to and perform calculations on most recent records. Upon the request, a new lambda function will retrieve

records from the previous 5 minutes and count how many people have entered and exited during those times. This information can be useful to calculate averages for the rates of people entering and exiting. This new information will then be published to a new SNS topic, *lambdaLink*, in which a second lambda message will use this information to put together a user-friendly message. Source code for this can be found in ***Appendix 3***.

This API will serve as a trigger for a Lambda function on the cloud. In short, every time that API is used, a certain piece of code can be executed requesting a response. The response is done through get, which sends the response in a JSON document for the web or phone application to display accordingly . In this setup, the data analysis component will begin and continue through the steps of delivering the proper response to the user.

Message to End-User

A user wants to obtain meaningful information about the state of the library rather than various variable values that might not mean anything to the user. A new lambda function will make use of the values obtained in the analysis section to put together a helpful message. The message will vary depending on the values obtained by the analysis section. For example, there can be times when a user will receive a message instructing him/her to avoid the library to study or times when the user will receive a message pointing out that the library is been somewhat empty recently.

The message will be published to a final SNS topic, linkEmail, to which a text message or email will be delivered containing the state of the library. Refer to ***Appendix 5*** for source code and the messages generated by the lambda function.

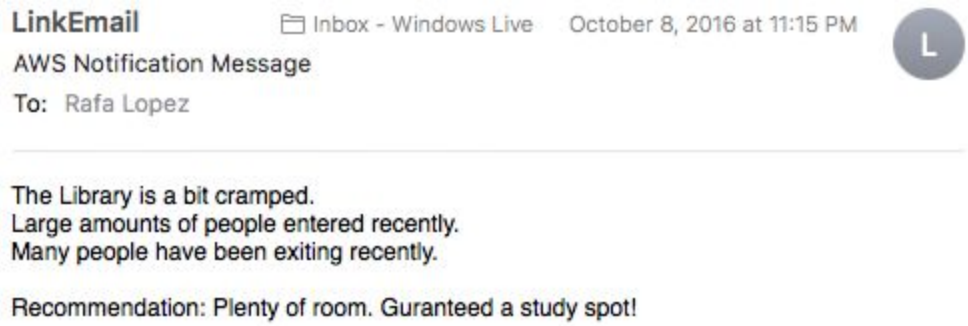


Figure 13. Email generated by AWS demonstrating the current state of the library.

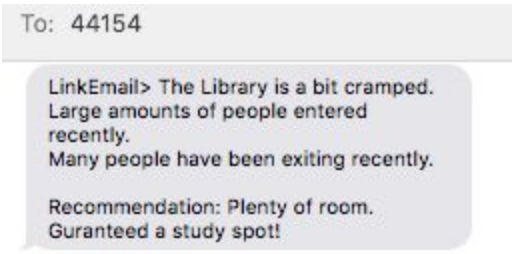


Figure 14. Text message generated by AWS demonstrating the current state of the library.

This concludes the setup used to tackle the library tracking application. This application served the purpose to demonstrate how a certain problem can be broken down into subsystems where a microcontroller, such as the Raspberry Pi, and a cloud service platform, AWS, can interact with one another and divide up the work. The difficult portion of the application was breaking down the application into subsystems as demonstrated by ***figure 12***. Each subsystem can be thought of a sub-problem that can be solved by a service in the cloud. From that same picture, the components in the cloud where composed of database services (DynamoDB), code execution services (Lambda), and push notification systems (SNS). All these are the building blocks that cloud platforms provide to users in order to develop their applications easier and faster.

Adding New Features to Existing Application

In the case of new implementations to arise in a certain application, it would only be required to figure out a way of choosing the right building block to accommodate for the new implementation. For example, this library tracking application can be useful to predict behavior in the library at specific days of the school year. One question a student or staff member may ask is how will the library look at 4pm on a particular Monday. This will require to learn patterns about the particular data obtained on Mondays. This can be performed by just querying all results in a database filtering out records that are on Monday at 4pm. This seems to require a Lambda function to perform that particular analysis on the DynamoDB database.

However, the data can vary depending on the Quarter and particular week of the quarter. A better approach would be creating a data model on all the data obtained aiming towards *machine learning*. AWS provides this service through its ***Machine Learning*** service: https://aws.amazon.com/machine-learning . A user can request a prediction on a set of values and obtained a response, i.e. on Monday at 4pm, expect about 120 people coming in per minute. Wiring these service to the current application will only require: creating a machine learning model through *machine learning*, creating a *lambda* function that requests prediction from the model created, setting up an *API gateway* to trigger the particular lambda function and obtain the response back.

Summary

These are just some concepts that can be applied within IoT by using a Raspberry Pi, Amazon Web Services and the Serverless framework. From this application described here are the services by AWS that were touched on:

1) Lambda: Code on the cloud
2) Simple Notification System (SNS): Delivering messages to devices subscribed to a topic.
3) DynamoDB: A NoSQL database, for fast deployment of a database.
4) API Gateway: A way for web/phone applications to request specific information from the cloud.
5) AWS IoT: Establish connection between external devices (i.e. microcontrollers) to the cloud.
6) Machine Learning: Create models about a particular dataset, and make predictions on sets of values. Predicting the future!

But there are more that can be easily incorporated to an application:

1) Elastic Compute Cloud (EC2): Provides scalable computing capacity in the cloud. Think powerful servers!
2) Simple Storage Service (S3): Provides storage for web applications. Storing and retrieving files on the cloud!
3) CodePipeline: Helpful to send updates and new code deployment to devices across the cloud. No need to update device one by one!
4) Redshift: SQL version database in AWS. Data Warehouse that is easy to configure and cost effective.
5) Athena: A good analytics tool that can help find patterns in data files. For example, retrieve all the error reports from logs generated by a particular system.
6) Rekognition: An artificial intelligence service that can perform analysis on images. For example, an application might want to organize pictures by their background. Rekognition can help differentiate between a picture in the beach and one in a forest.

These are just a few of the services that can be incorporated with AWS. In the case that a different cloud service is used, i.e. Microsoft Azure, a user should expect to find very similar services compared to these. The services and usage described were aimed to demonstrate what is out there in the cloud platform world. New services are bound to be created in the next following years, but the currently available ones serve as the backbone of cloud platforms.

APPENDICES

Appendix 1: Turning LED ON/OFF on button press

```
#include <wiringPi.h>

// Pin number declarations
const int pwmPin = 18; // PWM LED
const int ledPin = 23; // Regular LED
const int buttonPin = 17; // Active low button
const int pwmValue = 75; // used for LED brightness

int main(void) {
    // Setup part
    wiringPiSetupGpio(); // initializing wiringPi

    pinMode(pwmPin, PWM_OUTPUT); // set PWM LED as PWM output
    pinMode(ledPin, OUTPUT); // set regular LED as ouput
    pinMode(buttonPin, INPUT);
    pullUpDnControl(buttonPin, PUD_UP); // enable pull-up resistor on button

    // At this point the setup process is complete

    while(1) {
        if (digitalRead(buttonPin)) {
                pwmWrite(pwmPin, pwmValue); // write to LED bright setting
                digitalWrite(ledPin, LOW); // LED OFF
        }
        else {
                pwmWrite(pwmPin, 1024 - pwmValue); // PWM LED at dim setting
                digitalWrite(ledPin, HIGH); // LED ON
                delay(75);
                digitalWrite(ledPin, LOW); // TURN LED OFF
                delay(75);
        }
    }

    return 0;
}
```

Appendix 2: Creating a DynamoDB Table With Boto3 and Python

```
from __future__ import print_function
import boto3

import json
import logging

log = logging.getLogger()
log.setLevel(logging.DEBUG)

def handler(event, context):
    log.debug("Received event {}".format(json.dumps(event)))

    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.create_table (
        TableName='RAW_DATA',
        KeySchema=[
            {
                'AttributeName': 'id',
                'KeyType': 'HASH' # Sort Key
            },
            {
                'AttributeName': 'timestamp',
                'KeyType': 'RANGE' # Partition key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'id',
                'AttributeType': 'S'
            },
            {
                'AttributeName': 'timestamp',
                'AttributeType': 'S'
            },
        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 10,
            'WriteCapacityUnits': 10
        }
    )

    print("Table status:", table.table_status)

    return {}
```

Appendix 3: Inserting a new item into DynamoDB Table: RAW_DATA with message from SNS

```
from __future__ import print_function
import boto3
import decimal
import json
import logging

log = logging.getLogger()
log.setLevel(logging.DEBUG)

def handler(event, context):
        log.debug("Received event {}".format(json.dumps(event)))

        dynamodb = boto3.resource('dynamodb')
        table = dynamodb.Table('RAW_DATA')

        objStr = event['Records'][0]['Sns']['Message']
        obj = json.loads(objStr)

        table.put_item(
                Item={
                        'Id': 0:,
                        'timestamp': obj['timestamp'],
                        'location': obj['location'],
                        'in: obj['in'],
                        'out':obj['out'],
                }
        )

        print("Table status:", table.table_status)

        return {}
```

Appendix 4: Reading from DynamoDB and publishing to SNS Topic

```
from __future__ import print_function
import decimal
import time
import json
import boto3
import logging
from boto3.dynamodb.conditions import Key, Attr


log = logging.getLogger()
log.setLevel(logging.DEBUG)

def handler(event, context):
        log.debug("Received event {}".format(json.dumps(event)))
```

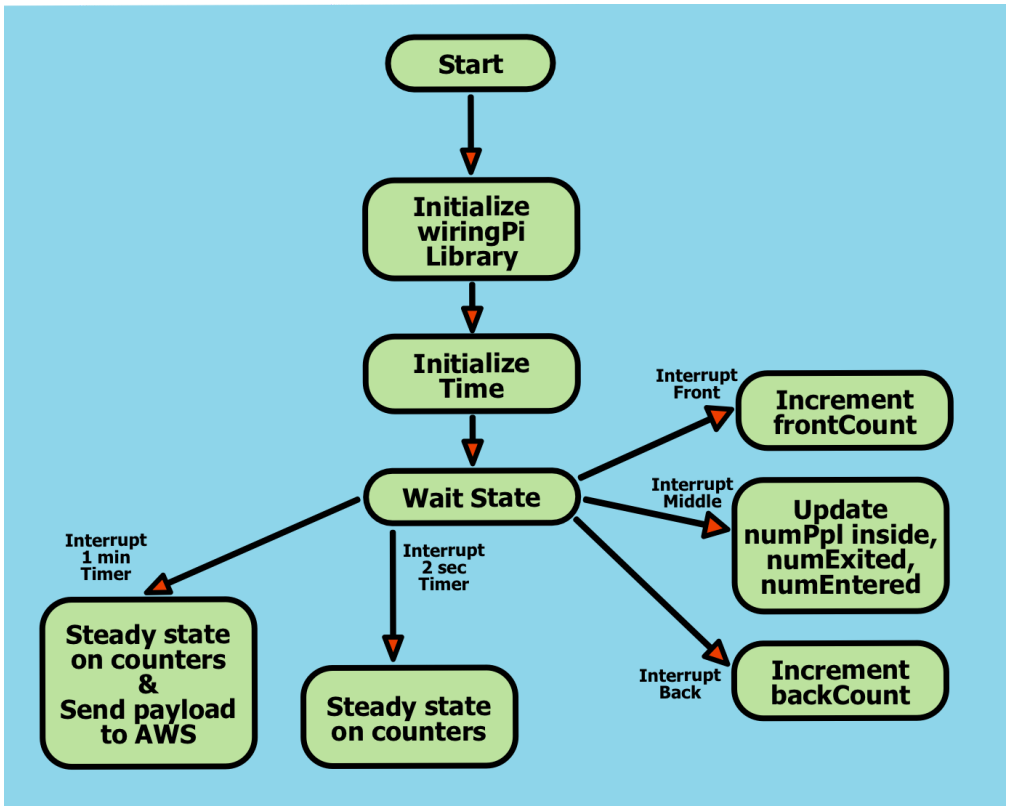
```
log.debug(event)
log.debug("CONTEXT")
log.debug(context)
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('RAW_DATA')
client = boto3.client('sns')


# This is dependent on the database having enough info for at least 5 minutes

current = str(int(time.time()) - (300)) #line 1

response = table.query(
        KeyConditionExpression = Key('id').eq('0') & Key('timestamp').gt(current)
)

totalIn = 0
totalOut = 0

for i in range(response['Count']):
item = response['Items'][i]
totalIn = totalIn + int(item['IN'])
totalOut = totalOut + int(item['OUT'])

ret = {
        "curStamp" : response['Items'][4]['timestamp'],
        "inRate" : str(totalIn / 5),
        "outRate" : str(totalOut / 5),
        "curCapacity" : "35"
}

publishResponse = client.publish(
        TargetArn = 'arn:aws:sns:us-east-1:<AWS Account Number>:LambdaLink',
        Message = json.dumps({'default': json.dumps(ret)}),
        MessageStructure = 'json'
)

return ret, publishResponse
```

Appendix 5: Generating meaningful messages from analysis and publishing to SMS/Email

```
from __future__ import print_function

import json
import logging
import boto3

log = logging.getLogger()
log.setLevel(logging.DEBUG)

def handler(event, context):

        log.debug("Received event {}".format(json.dumps(event)))

        # keep
        client = boto3.client('sns')

        objStr = event['Records'][0]['Sns']['Message']

        outputMsg = ""
        obj = json.loads(objStr)

        log.debug(obj)
        log.debug(obj['curCapacity'])

        # Setting the message of the current capacity of the library
        # based on the percentage of 'curCapacity'
        if (int(obj['curCapacity'])  < 25):
        outputMsg += 'The Library is relative empty.'
        elif (int(obj['curCapacity']) < 50):
        outputMsg += 'The Library is a bit cramped.'
        elif (int(obj['curCapacity']) < 75):
        outputMsg += 'The Library is semi-cramped.'
        elif (int(obj['curCapacity']) < 85):
        outputMsg += 'The Library is packed.'
        else:
        outputMsg += 'The Library is packed.'

        outputMsg += ' \n'
        # Setting the message of the traffic of people coming in
        if (int(obj['inRate']) == 0):
        outputMsg += 'No one has been entering recently.'
        elif (int(obj['inRate']) < 10):
        outputMsg += 'A few people have entered recently.'
        elif (int(obj['inRate']) < 20):
        outputMsg += 'A moderate amount of people entered recently.'
        elif (int(obj['inRate']) < 30):
        outputMsg += 'Many people have been entering recently.'
        else:
        outputMsg += 'Large amounts of people entered recently.'
```

```
outputMsg += ' \n'

# Setting the message of the traffic of people coming out
if (int(obj['outRate']) == 0):
outputMsg += 'No one has been exiting recently. '
elif (int(obj['outRate']) < 10):
outputMsg += 'A few people have exited recently. '
elif (int(obj['outRate']) < 20):
outputMsg += 'A moderate amount of people exited recently. '
elif (int(obj['outRate']) < 30):
outputMsg += 'Many people have been exiting recently. '
else:
outputMsg += 'Large amounts of people exited recently. '


# Setting the reccomendations messages
outputMsg += '\n\nRecommendation: '
if (int(obj['curCapacity'])  < 25):
outputMsg += 'Lots of space to study. Enough time to grab a drink/snack!'
elif (int(obj['curCapacity']) < 50):
outputMsg += 'Plenty of room. Guaranteed a study spot!'
elif (int(obj['curCapacity']) < 75):
outputMsg += 'Spots are available. Hurry before they are taken!'
elif (int(obj['curCapacity']) < 85):
outputMsg += 'Might be difficult finding a spot. Make sure to check each floor!'
Else:
outputMsg += 'Everything is mostly taken. Consider another place to study around
campus. UU, Dexter Lawn or an open classroom are great options!'


publishResponse = client.publish(
        TargetArn = 'arn:aws:sns:us-east-1:<AWS Account number>:LinkEmail',
        Message = outputMsg
)

log.debug("Here is the response after publishing")
log.debug(publishResponse)
return publishResponse
```

Appendix 6: Algorithm flow for line tracking using 3 motion sensors.



Appendix 7: Source code for line tracking algorithm using 3 motion sensors.

```
// Author: Rafael Lopez
// This piece of code uses the Raspberry Pi along with the wiringPi library
// to connect with 3 motion sensors to detect
//  when a person enters or exists in a straight line.
//  A successful entrance triggers the sensors in this order: front -> middle -> back
//  A successful exit triggers the sensors in this order: back -> middle -> front
// Errors can occur, and the timer interrupt attempts to resets the state to a stable state
// in case any discrepancies have occurred.
// At every timer interrupt something meaningful can be done:
// i.e. print total people inside, and have exited and entered.


#include <stdio.h>
#include <stdlib.h>
#include <wiringPi.h>
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>

// Pin numbers of GPIOS for Motion sensors
static const int front = 17;
static const int middle = 18;
static const int back = 22;

// Values keeping track how many times each sensor
// has been triggered
static volatile int16_t frontVal = 0;
static volatile int16_t middleVal = 0;
static volatile int16_t backVal = 0;
static volatile int16_t current = 0;
static volatile int16_t in = 0;
static volatile int16_t out = 0;

// ISRS to handle detection on buttons
void ISR_front(void);
void ISR_middle(void);
void ISR_back(void);

// Functions to take care of interrupts at every second
void timeInterrupt(void);
void SetTimer(struct itimerval *timer);

// Interval to generate a timer interrupt
#define INTERVAL 1000

int main(void)
{
    // Setup gpios in Pi
    wiringPiSetupGpio();
```

```
    // Set pins as inputs
    pinMode(front, INPUT);
    pinMode(middle, iNPUT);
    pinMode(back, INPUT);


    // Set Pull up resistors
    pullUpDnControl(front, PUD_UP);
    pullUpDnControl(middle, PUD_UP);
    pullUpDnControl(back, PUD_UP);

    // Setting ISRs on each input
    wiringPiISR(front, INT_EDGE_RISING, &ISR_front);
    wiringPiISR(middle, INT_EDGE_RISING, &ISR_middle);
    wiringPiISR(back, INT_EDGE_RISING, &ISR_back);


    // timer alarm configuration
    struct itimerval timer;

    // Sets up timer, just call this
    //make sure INTERVAL is set to the number of milliseconds desired
    SetTimer(&timer);

    // LOOP FOR-E-V-E-R
    while(1)
        ;

    return 0;
}

void ISR_front(void) {
    frontVal++;
}

void ISR_middle(void) {
    if (frontVal > backVal) {
        in++;
        current++;
    }

    if (backVal > frontVal) {
        out++;
        current--;
    }

    if (current < 0)
        current = 0;
}
```

```
void ISR_back(void) {
    backVal++;
}

void SetTimer(itimerval *timer) {
    // Choose the interrupt function to act on when the timer goes off
    // Everytime an alarm goes off, updateSeconds() is called
    if (signal(SIGALRM, (void (*)(int)) timeInterrupt) == SIG_ERR) {
                    perror("Unable to catch SIGALRM");
                    exit(1);
        }

        timer->it_value.tv_sec = INTERVAL/1000;
        timer->it_value.tv_usec = (INTERVAL * 1000) % 1000000;
        timer->it_interval = timer->it_value;

        if (setitimer(ITIMER_REAL, timer, NULL) == -1) {
                    perror("ERROR calling setitimer(), exiting...");
                    exit(1);
        }
}


void timeInterrupt(void) {
    if (frontVal != backVal) {
        if (frontVal > backVal)
               backVal = frontVal;
        else
               frontVal = backVal;
    }

    // DO SOMETHING MEANINGFUL HERE!!!
    backVal = frontVal = 0;
    in = out = 0;

}
```

Appendix 8: Sending number of button presses to AWS via MQTT

```
/*
 * Copyright 2010-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *  http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

/**
 * @file subscribe_publish_library_sample.c
 * @brief simple MQTT publish and subscribe on the same topic using the SDK as a library
 *
 * This example takes the parameters from the aws_iot_config.h file and establishes a
connection to the AWS IoT MQTT Platform.
 * It publishes to the topic - "mqttTopic"
 *
 * If all the certs are correct, you should see the messages received by the application in a
loop.
 *
 * The application takes in the certificate path, host name , port and the number of times the
publish should happen.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <limits.h>
#include <string.h>

#include <wiringPi.h>

#include "aws_iot_config.h"
#include "aws_iot_log.h"
#include "aws_iot_version.h"
#include "aws_iot_mqtt_client_interface.h"

/**
 * @brief Default cert location
 */
char certDirectory[PATH_MAX + 1] = "../../../certs";
```

```
/**
 * @brief Default MQTT HOST URL is pulled from the aws_iot_config.h
 */
char HostAddress[255] = AWS_IOT_MQTT_HOST;

/**
 * @brief Default MQTT port is pulled from the aws_iot_config.h
 */
uint32_t port = AWS_IOT_MQTT_PORT;

/**
 * @brief This parameter will avoid infinite loop of publish and exit the program after
certain number of publishes
 */
uint32_t publishCount = 0;

void iot_subscribe_callback_handler(AWS_IoT_Client *pClient, char *topicName, uint16_t
topicNameLen,
                                    IoT_Publish_Message_Params *params,
void *pData) {
    IOT_UNUSED(pData);
    IOT_UNUSED(pClient);
    IOT_INFO("Subscribe callback");
    IOT_INFO("%.*s\t%.*s", topicNameLen, topicName, (int) params->payloadLen,
params->payload);
}

void disconnectCallbackHandler(AWS_IoT_Client *pClient, void *data) {
    IOT_WARN("MQTT Disconnect");
    IoT_Error_t rc = FAILURE;

    if(NULL == pClient) {
        return;
    }

    IOT_UNUSED(data);

    if(aws_iot_is_autoreconnect_enabled(pClient)) {
        IOT_INFO("Auto Reconnect is enabled, Reconnecting attempt will start now");
    } else {
        IOT_WARN("Auto Reconnect not enabled. Starting manual reconnect...");
        rc = aws_iot_mqtt_attempt_reconnect(pClient);
        if(NETWORK_RECONNECTED == rc) {
                IOT_WARN("Manual Reconnect Successful");
        } else {
                IOT_WARN("Manual Reconnect Failed - %d", rc);
        }
    }
}
```

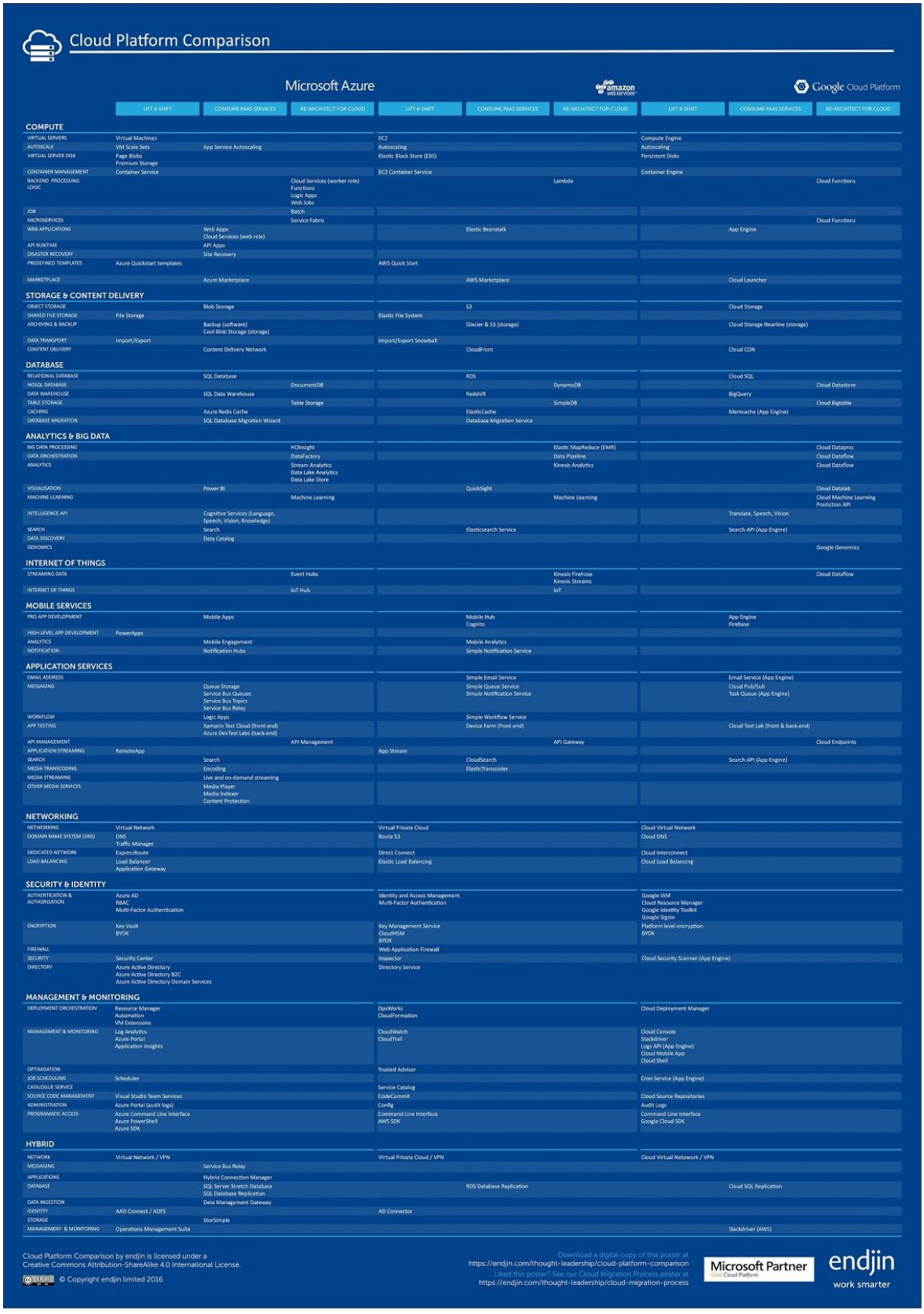
```
void parseInputArgsForConnectParams(int argc, char **argv) {
    int opt;

    while(-1 != (opt = getopt(argc, argv, "h:p:c:x:"))) {
        switch(opt) {
                case 'h':
                        strcpy(HostAddress, optarg);
                        IOT_DEBUG("Host %s", optarg);
                        break;
                case 'p':
                        port = atoi(optarg);
                        IOT_DEBUG("arg %s", optarg);
                        break;
                case 'c':
                        strcpy(certDirectory, optarg);
                        IOT_DEBUG("cert root directory %s", optarg);
                        break;
                case 'x':
                        publishCount = atoi(optarg);
                        IOT_DEBUG("publish %s times\n", optarg);
                        break;
                case '?':
                        if(optopt == 'c') {
                                IOT_ERROR("Option -%c requires an argument.", optopt);
                        } else if(isprint(optopt)) {
                                IOT_WARN("Unknown option `-%c'.", optopt);
                        } else {
                                IOT_WARN("Unknown option character `\\x%x'.", optopt);
                        }
                        break;
                default:
                        IOT_ERROR("Error in command line argument parsing");
                        break;
        }
    }

}

// Pin number declarations
// -------------
const int pwmPin      = 18; // PWM LED
const int ledPin      = 23; // Regular LED
const int buttonPin   = 17; // Active low button
const int pwmValue    = 75; // Used for LED Brightnes: mid level
// -------------


int main(int argc, char **argv) {
    bool infinitePublishFlag = true;
```

```
    char rootCA[PATH_MAX + 1];
    char clientCRT[PATH_MAX + 1];
    char clientKey[PATH_MAX + 1];
    char CurrentWD[PATH_MAX + 1];
    char cPayload[100];

    int32_t i = 0;
    int32_t isPressed = 0;

    IoT_Error_t rc = FAILURE;

    AWS_IoT_Client client;
    IoT_Client_Init_Params mqttInitParams = iotClientInitParamsDefault;
    IoT_Client_Connect_Params connectParams = iotClientConnectParamsDefault;

    IoT_Publish_Message_Params paramsQOS0;


    // Code to Initialize Pi, and GPIOs
    // -------------
    wiringPiSetupGpio(); // initializing wiringPi

    pinMode(pwmPin, PWM_OUTPUT); // set PWM LED as PWM output
    pinMode(ledPin, OUTPUT); // set regular LED as ouput
    pinMode(buttonPin, INPUT);
    pullUpDnControl(buttonPin, PUD_UP); // enable pull-up resistor on button
    // -------------


    parseInputArgsForConnectParams(argc, argv);

    IOT_INFO("\nAWS IoT SDK Version %d.%d.%d-%s\n", VERSION_MAJOR, VERSION_MINOR,
VERSION_PATCH, VERSION_TAG);

    getcwd(CurrentWD, sizeof(CurrentWD));
    snprintf(rootCA, PATH_MAX + 1, "%s/%s/%s", CurrentWD, certDirectory,
AWS_IOT_ROOT_CA_FILENAME);
    snprintf(clientCRT, PATH_MAX + 1, "%s/%s/%s", CurrentWD, certDirectory,
AWS_IOT_CERTIFICATE_FILENAME);
    snprintf(clientKey, PATH_MAX + 1, "%s/%s/%s", CurrentWD, certDirectory,
AWS_IOT_PRIVATE_KEY_FILENAME);

    IOT_DEBUG("rootCA %s", rootCA);
    IOT_DEBUG("clientCRT %s", clientCRT);
    IOT_DEBUG("clientKey %s", clientKey);
    mqttInitParams.enableAutoReconnect = false; // We enable this later below
    mqttInitParams.pHostURL = HostAddress;
    mqttInitParams.port = port;
    mqttInitParams.pRootCALocation = rootCA;
    mqttInitParams.pDeviceCertLocation = clientCRT;
```

```
    mqttInitParams.pDevicePrivateKeyLocation = clientKey;
    mqttInitParams.mqttCommandTimeout_ms = 20000;
    mqttInitParams.tlsHandshakeTimeout_ms = 5000;
    mqttInitParams.isSSLHostnameVerify = true;
    mqttInitParams.disconnectHandler = disconnectCallbackHandler;
    mqttInitParams.disconnectHandlerData = NULL;

    rc = aws_iot_mqtt_init(&client, &mqttInitParams);
    if(SUCCESS != rc) {
        IOT_ERROR("aws_iot_mqtt_init returned error : %d ", rc);
        return rc;
    }

    connectParams.keepAliveIntervalInSec = 10;
    connectParams.isCleanSession = true;
    connectParams.MQTTVersion = MQTT_3_1_1;
    connectParams.pClientID = AWS_IOT_MQTT_CLIENT_ID;
    connectParams.clientIDLen = (uint16_t) strlen(AWS_IOT_MQTT_CLIENT_ID);
    connectParams.isWillMsgPresent = false;

    IOT_INFO("Connecting...");
    rc = aws_iot_mqtt_connect(&client, &connectParams);
    if(SUCCESS != rc) {
        IOT_ERROR("Error(%d) connecting to %s:%d", rc, mqttInitParams.pHostURL,
mqttInitParams.port);
        return rc;
    }
    /*
     * Enable Auto Reconnect functionality. Minimum and Maximum time of Exponential backoff
are set in aws_iot_config.h
     *  #AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL
     *  #AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL
     */
    rc = aws_iot_mqtt_autoreconnect_set_status(&client, true);
    if(SUCCESS != rc) {
        IOT_ERROR("Unable to set Auto Reconnect to true - %d", rc);
        return rc;
    }

    paramsQOS0.qos = QOS0;
    paramsQOS0.payload = (void *) cPayload;
    paramsQOS0.isRetained = 0;

    if(publishCount != 0) {
        infinitePublishFlag = false;
    }

    while((NETWORK_ATTEMPTING_RECONNECT == rc || NETWORK_RECONNECTED == rc || SUCCESS == rc)
          && (publishCount > 0 || infinitePublishFlag)) {
```

```
        //Max time the yield function will wait for read messages
        rc = aws_iot_mqtt_yield(&client, 100);
        if(NETWORK_ATTEMPTING_RECONNECT == rc) {
                // If the client is attempting to reconnect we will skip the rest of the loop.
                continue;
        }

        if(digitalRead(buttonPin)) {
                if (!isPressed) { // actually recognized button
                        isPressed = 1;
                        pwmWrite(pwmPin, pwmValue); // write to LED bright setting
                        digitalWrite(ledPin, LOW); // LED OFF

                        sprintf(cPayload, "%s presses: %d", "Hello World!", i++);
                        paramsQOS0.payloadLen = strlen(cPayload);
                        IOT_INFO("--> Publishing to AWS Topic: mqttTopic");
                        rc = aws_iot_mqtt_publish(&client, "mqttTopic", 11, &paramsQOS0);
                }
        }
        else {
                if (isPressed) {
                        isPressed = 0;
                        pwmWrite(pwmPin, 1024 - pwmValue); // set pwm value
                        digitalWrite(ledPin, LOW); // Turn LED OFF
                }
        }
    }

    if(SUCCESS != rc) {
        IOT_ERROR("An error occurred in the loop.\n");
    } else {
        IOT_INFO("Publish done\n");
    }
    return rc;
}
```

Appendix 9: Comparison Matrix for Big Three Cloud Platforms (pdf is in drive!)



Cloud Platform Comparison

	Microsoft Azure			amazon web services			Google Cloud Platform		
	LIFT & SHIFT	CONSUME PAAS SERVICES	RE-ARCHITECT FOR CLOUD	LIFT & SHIFT	CONSUME PAAS SERVICES	RE-ARCHITECT FOR CLOUD	LIFT & SHIFT	CONSUME PAAS SERVICES	RE-ARCHITECT FOR CLOUD
COMPUTE									
VIRTUAL SERVERS	Virtual Machines			EC2			Compute Engine		
AUTOSCALE	VM Scale Sets	App Service Autoscaling		Autoscaling			Autoscaling		
VIRTUAL SERVER DISK	Page Blobs Premium Storage			Elastic Block Store (EBS)			Persistent Disks		
CONTAINER MANAGEMENT	Container Service			EC2 Container Service			Container Engine		
BACKEND PROCESSING LOGIC			Cloud Services (worker role) Functions Logic Apps Web Jobs			Lambda			Cloud Functions
JOB			Batch						
MICROSERVICES			Service Fabric						Cloud Functions
WEB APPLICATIONS		Web Apps Cloud Services (web role)			Elastic Beanstalk			App Engine	
API RUNTIME		API Apps							
DISASTER RECOVERY		Site Recovery							
PREDEFINED TEMPLATES	Azure Quickstart templates			AWS Quick Start					
MARKETPLACE		Azure Marketplace			AWS Marketplace			Cloud Launcher	
STORAGE & CONTENT DELIVERY									
OBJECT STORAGE		Blob Storage			S3			Cloud Storage	
SHARED FILE STORAGE	File Storage			Elastic File System					
ARCHIVING & BACKUP		Backup (software) Cool Blob Storage (storage)			Glacier & S3 (storage)			Cloud Storage Nearline (storage)	
DATA TRANSPORT	Import/Export			Import/Export Snowball					
CONTENT DELIVERY		Content Delivery Network			CloudFront			Cloud CDN	
DATABASE									
RELATIONAL DATABASE		SQL Database			RDS			Cloud SQL	
NOSQL DATABASE			DocumentDB			DynamoDB			Cloud Datastore
DATA WAREHOUSE		SQL Data Warehouse			Redshift			BigQuery	
TABLE STORAGE			Table Storage			SimpleDB			Cloud Bigtable
CACHING		Azure Redis Cache			ElastiCache			Memcache (App Engine)	
DATABASE MIGRATION		SQL Database Migration Wizard			Database Migration Service				
ANALYTICS & BIG DATA									
BIG DATA PROCESSING			HDInsight			Elastic MapReduce (EMR)			Cloud Dataproc
DATA ORCHESTRATION			DataFactory			Data Pipeline			Cloud Dataflow
ANALYTICS			Stream Analytics Data Lake Analytics Data Lake Store			Kinesis Analytics			Cloud Dataflow
VISUALISATION		Power BI			QuickSight				Cloud Datalab
MACHINE LEARNING			Machine Learning			Machine Learning			Cloud Machine Learning Prediction API
INTELLIGENCE API		Cognitive Services (Language, Speech, Vision, Knowledge)						Translate, Speech, Vision	
SEARCH		Search			Elasticsearch Service			Search API (App Engine)	
DATA DISCOVERY		Data Catalog							
GENOMICS									Google Genomics
INTERNET OF THINGS									
STREAMING DATA			Event Hubs			Kinesis Firehose Kinesis Streams			Cloud Dataflow
INTERNET OF THINGS			IoT Hub			IoT			
MOBILE SERVICES									
PRO APP DEVELOPMENT		Mobile Apps			Mobile Hub Cognito			App Engine Firebase	
HIGH-LEVEL APP DEVELOPMENT	PowerApps								
ANALYTICS		Mobile Engagement			Mobile Analytics				
NOTIFICATION		Notification Hubs			Simple Notification Service				
APPLICATION SERVICES									
EMAIL ADDRESS					Simple Email Service			Email Service (App Engine)	
MESSAGING		Queue Storage Service Bus Queues Service Bus Topics Service Bus Relay			Simple Queue Service Simple Notification Service			Cloud Pub/Sub Task Queue (App Engine)	
WORKFLOW		Logic Apps			Simple Workflow Service				
APP TESTING		Xamarin Test Cloud (front end) Azure DevTest Labs (back-end)			Device Farm (front end)			Cloud Test Lab (front & back-end)	
API MANAGEMENT			API Management			API Gateway			Cloud Endpoints
APPLICATION STREAMING	RemoteApp			App Stream					
SEARCH		Search			CloudSearch			Search API (App Engine)	
MEDIA TRANSCODING		Encoding			ElasticTranscoder				
MEDIA STREAMING		Live and on-demand streaming							
OTHER MEDIA SERVICES		Media Player Media Indexer Content Protection							
NETWORKING									
NETWORKING	Virtual Network			Virtual Private Cloud			Cloud Virtual Network		
DOMAIN NAME SYSTEM (DNS)	DNS Traffic Manager			Route 53			Cloud DNS		
DEDICATED NETWORK	ExpressRoute			Direct Connect			Cloud Interconnect		
LOAD BALANCING	Load Balancer Application Gateway			Elastic Load Balancing			Cloud Load Balancing		
SECURITY & IDENTITY									
AUTHENTICATION & AUTHORISATION	Azure AD RBAC Multi-Factor Authentication			Identity and Access Management Multi-Factor Authentication			Google IAM Cloud Resource Manager Google Identity Toolkit Google Signin		
ENCRYPTION	Key Vault BYOK			Key Management Service CloudHSM BYOK			Platform level encryption BYOK		
FIREWALL				Web Application Firewall					
SECURITY	Security Center			Inspector			Cloud Security Scanner (App Engine)		
DIRECTORY	Azure Active Directory Azure Active Directory B2C Azure Active Directory Domain Services			Directory Service					
MANAGEMENT & MONITORING									
DEPLOYMENT ORCHESTRATION	Resource Manager Automation VM Extensions			OpsWorks CloudFormation			Cloud Deployment Manager		
MANAGEMENT & MONITORING	Log Analytics Azure Portal Application Insights			CloudWatch CloudTrail			Cloud Console Stackdriver Logs API (App Engine) Cloud Mobile App Cloud Shell		
OPTIMISATION				Trusted Advisor					
JOB SCHEDULING	Scheduler						Cron Service (App Engine)		
CATALOGUE SERVICE				Service Catalog					
SOURCE CODE MANAGEMENT	Visual Studio Team Services			CodeCommit			Cloud Source Repositories		
ADMINISTRATION	Azure Portal (audit logs)			Config			Audit Logs		
PROGRAMATIC ACCESS	Azure Command Line Interface Azure PowerShell Azure SDK			Command Line Interface AWS SDK			Command Line Interface Google Cloud SDK		
HYBRID									
NETWORK	Virtual Network / VPN			Virtual Private Cloud / VPN			Cloud Virtual Network / VPN		
MESSAGING		Service Bus Relay							
APPLICATIONS		Hybrid Connection Manager							
DATABASE		SQL Server Stretch Database SQL Database Replication			RDS Database Replication			Cloud SQL Replication	
DATA INGESTION		Data Management Gateway							
IDENTITY	AAD Connect / ADFS			AD Connector					
STORAGE		StorSimple							
MANAGEMENT & MONITORING	Operations Management Suite							Slackdriver (AWS)	

Download a digital copy of this poster at
https://endjin.com/thought-leadership/cloud-platform-comparison
Liked this poster? See our Cloud Migration Process poster at
https://endjin.com/thought-leadership/cloud-migration-process

Microsoft Partner
Gold Cloud Platform

endjin
work smarter