UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Spring Report

Project: Docker Containers Deployed Using Bluemix

Name: Thomas Flynn

ID: 16117743

Supervisor: Dr. Sean McGrath

Course: Information & Network Security MEng

Year: 2017

Department: Electronic & Computer Engineering

Table of Contents

Illustration Index

1 Introduction and Report Outline

1.1 Project Description

Docker Containers remove unnecessary resources out of the operating system. This results in increased CPU, memory, and network resources for middleware and application execution. As a general rule of thumb, Containers are 10 times more efficient than virtual machines in terms of CPU, memory, network and other resource utilization.

IBM has formed a close partnership with Docker in order to make Containers Enterprise Ready. IBM Software products such as Websphere Application Server (WAS) and IBM BlueMix are already Container ready.

This project is based on open source standards and is aimed at engineering a scalable back-end solution for tracking in real-time the number of vehicles and students on the university campus. Due to their scalable capabilities, Containers were chosen in order to handle the variation of traffic on campus over a 24 hour period. During periods of peak traffic, the number of Containers handling the processing of sensor data can efficiently scale up to meet demand. Computing resources will not be wasted during periods of low traffic because the number of Containers will be automatically scaled down.

The clients (cars, students) will transmit GPS (Global Position System) data periodically to the server using a lightweight IoT protocol known as MQTT (MQ Telemetry Transport). This is a commonly used protocol for constrained devices. Both client and server will incorporate features to ensure that communication and caching of data is only performed when appropriate. For example the client will not attempt to transmit data if its coordinates are outside college bounds, similarly the server won't cache data received that is outside college bounds.

IBM Cloudant is a managed NoSQL JSON database service built to ensure that the flow of data between an application and its database remains uninterrupted and highly performant. Cloudant has excellent features for indexing, quering and visualizing geo-spatial data.

InfluxDB is used to organize the data from the sensors into a time-series for efficiency. OpenWhisk actions will be triggered to organize the data geo-spatially from Cloudant before feeding it into the time-series database.

Node-RED is a simple visual tool that makes it easy to wire together events and devices for the Internet of Things. To test the scalability of the application, Node-Red flows will inject messages to the MQTT topics "car-iotp" and "mobile-iotp" in order to determine the load balancing benchmarks of the project.

In order to help future students build applications on top of this project, various Development Operations aspects will be considered such as GitHub repositories, continuous integration/deployment pipelines, Container orchestration and microservices frameworks such as Amalgam8. This area of the project aims to reduce the operational complexity for future students who want to develop scalable Smart Campus Applications using microservices.

1.2 Report Outline

1.2.1 Overall Outline

The main objective of this report is to provide future students with sufficient documentation in order to help them with the initial learning curve when attempting to build on top of an existing project such as this one.

Progress on the project will be halted from March 15th until May1st. This time will be used to focus on studying for exams. The report will therefore also assist with refreshing memory when taking up work on the project after the hiatus.

1.2.2 Literature Survey

Discusses the initial research for the chosen technologies.

1.2.3 Architecture

Discusses the interaction of the various components in more detail.

1.2.4 Theory

Discusses the work completed in the Spring semesters.

1.2.5 Project Management

Discusses the project management strategy.

1.2.6 Progress to Date

Discusses the progress made in each week of the Spring semester. It heavily references the repository found on GitHub that contains all of the theses documentation.[1]

1.2.7 Evaluation

Discusses the limitations and problems that will be encountered during the Summer semester.

1.2.8 Bibliography

Listing of references used.

2 Literature Survey

2.1 Section Outline

This section gives an overview of the content researched during the Spring semester. It starts with a high level discussion of IoT pub/sub architecture. This will provide the reader with a perspective on both the design and scope of the project.

2.1.1 Section Layout

- IoT pub/sub Architecture discussion
- Load balancing using HAProxy discussion
- MQTT brokers discussion
- Redis and Kafka discussion
- **E**lastisearch, **L**ogstash, **K**ibana stack discussion
- IBM's platform logging and monitoring solution discussion
- A discussion on Container orchestration and the Docker compose tool
- A discussion on the problems associated with service discovery
- Amalgam8 microservices framework discussion
- DevOps CI/CD pipeline discussion
- InfluxDB discussion
- MQTT and REST discussion
- A discussion on similar projects

2.2 IoT Architecture

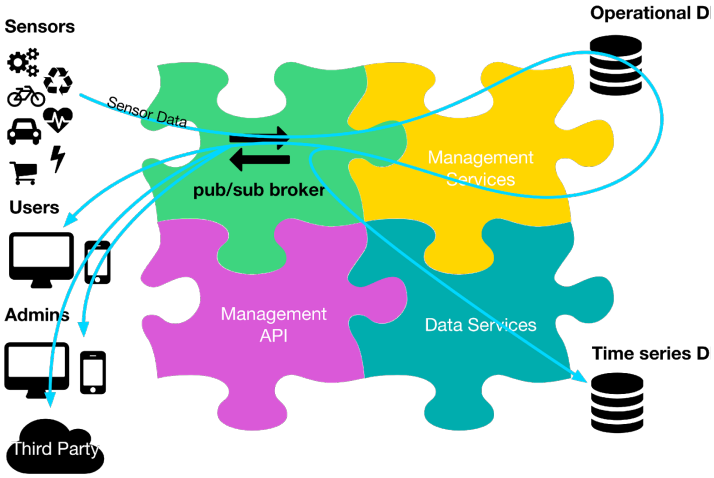

Illustration 1: IoT pub/sub data

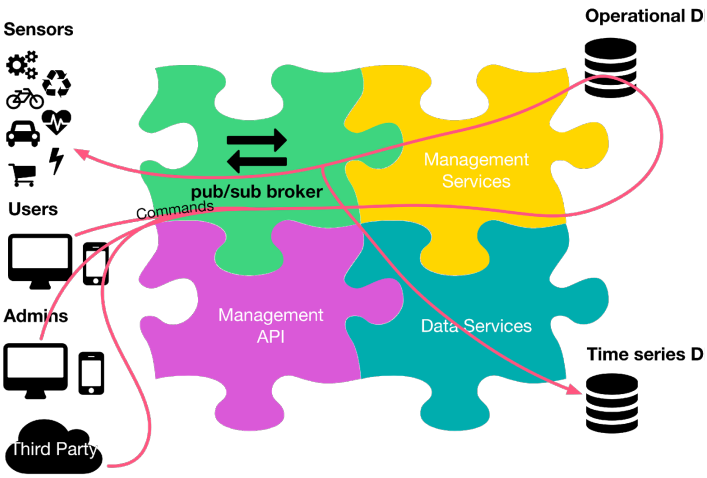

Illustration 2: IoT pub/sub commands

The main idea of the above architecture is to expose a pub/sub broker up front to the users and the devices. This design makes it is possible to direct the interaction between the users and the sensors, both for commands and data. This approach reduces latency, and allows for offline handling of time-series data and analytics.[2]

2.2.1 Sensor API

A sensor API, which is called by the sensors to deliver data readings and receive commands.

2.2.2 Public API

A public API, which is called by the sensors to retrieve real-time data, historical data, and to manage the devices.

2.2.2 Operations Services

A set of operational services, which are responsible for authentication and authorization, among other things; these services manage their own database.

2.2.3 Data Services

A set of data services, which are responsible for storing and analyzing the data, either in real time or offline.

2.3 Load Balancing

2.3.1 HAProxy

HAProxy is an open source software load balancer for both layer four and layer seven. Written in C and runing on top of Linux, HAProxy uses an event-driven model to reduce memory usage and gain high performance.[3] HAProxy also uses a single-buffering technique to reduce cost memory copy.

Illustration 3: HAProxy Logo

Although HAProxy is high performance, its design goal is to handle a large number of concurrent connections. It can only load balance traffic at the flow level and is unable to vertically divide a single elephant TCP flow into multiple mice flows.[4] HAProxy creates a session for each connection. A session consists of two TCP connections, one from the client to the load balancer and one from the load balancer to the server. The user needs to specify the load balancing policy in a config file before starting HAProxy.

2.3.2 Reasons for using MQTT over Websockets

- Connect directly web applications and things, so you can bootstrap a whole application with very little backend.
- Any other applications that don't want to use the 1883/8883 port and want to go over HTTP/HTTPS instead - this could be so that there is less of a chance of being blocked by a firewall, as most firewalls will let HTTP traffic through

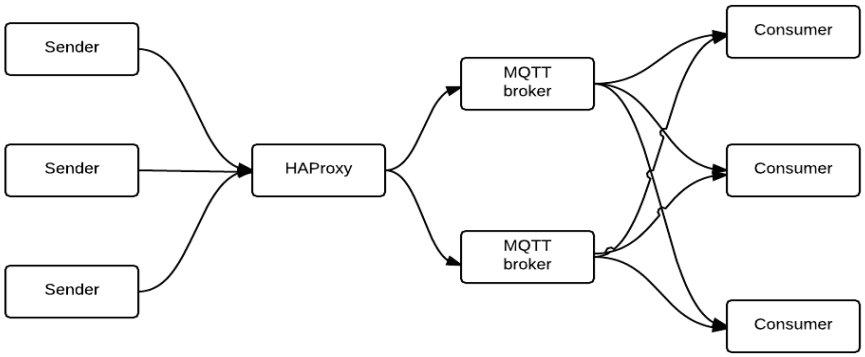

Illustration 4: HAProxy Broker Diagram

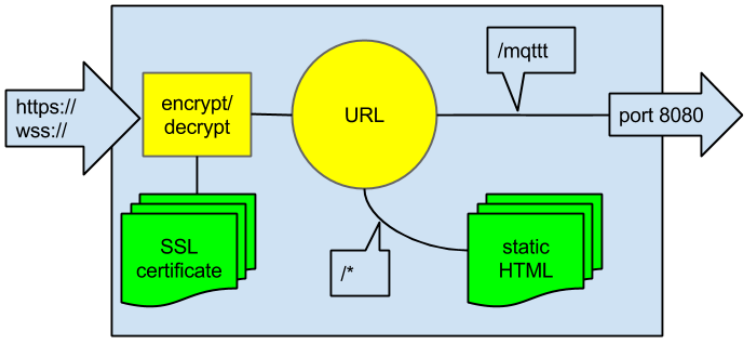

Illustration 5: HAProxy SSL Decryption

2.4 MQTT Brokers and Clients

2.4.1 Mosca

Mosca[5] sits between your system and the devices. It is open source and written in Javascript. Mosca is embeddable within any node application, allowing complete customization and the implementation of the pub-sub architecture.

Illustration 6: Mosca Logo

2.4.1.1 Mosca Features

- MQTT 3.1 and 3.1.1 compliant.
- QoS 0 and QoS 1.
- Various storage options for QoS 1 offline packets, and subscriptions.
- Usable inside any other Node.js app.

2.4.2 RabbitMQ

RabbitMQ[6] is a messaging broker - an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

Illustration 7: RabbitMQ Logo

2.4.2.1 RabbitMQ Features

- Persistence, delivery acknowledgements, publisher confirms, and high availability.
- Several RabbitMQ servers on a local network can be clustered together, forming a single logical broker.
- RabbitMQ supports messaging over a variety of messaging protocols.

2.4.3 MQTT.js Client Library

MQTT.js[7] is a client library for the MQTT protocol, written in JavaScript for node.js and the browser. MQTT.js is an OPEN Open Source Project. v2.0.0 removes support for node v0.8, v0.10 and v0.12, and it is 3x faster in sending packets. The new Client improves performance by a 30% factor, embeds Websocket support, and has better support for QoS 1 and 2.

Illustration 8: MQTT.js Logo

2.5 Redis

Redis[8] is an open source, BSD licensed, advanced key-value cache and store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, sorted sets, bitmaps and hyperlogs.

Illustration 9: Redis Logo

Redis is a bit different from Kafka in terms of its storage and various functionalities. At its core, Redis is an in-memory data store that can be used as a high-performance database, a cache, and a message broker. It is perfect for real-time data processing.

Redis also has various clients written in several languages which can be used to write custom programs for the insertion and retrieval of data.[9] This is an advantage over Kafka since Kafka only has a Java client. The main similarity between the two is that they both provide a messaging service. But for the purpose of log aggregation, the various data structures of Redis do it more efficiently.

2.6 Kafka

Apache Kafka[10] is a distributed publish-subscribe messaging system. It is designed to support the following Persistent messaging with O(1) disk structures that provide constant time performance even with many TB of stored messages.

Illustration 10: Kafka Logo

High-throughput: even with very modest hardware Kafka can support hundreds of thousands of messages per second. Explicit support for partitioning messages over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics. Kafka provides parallelism in processing. More than one consumer from a consumer group can retrieve data simultaneously, in the same order that messages are stored.

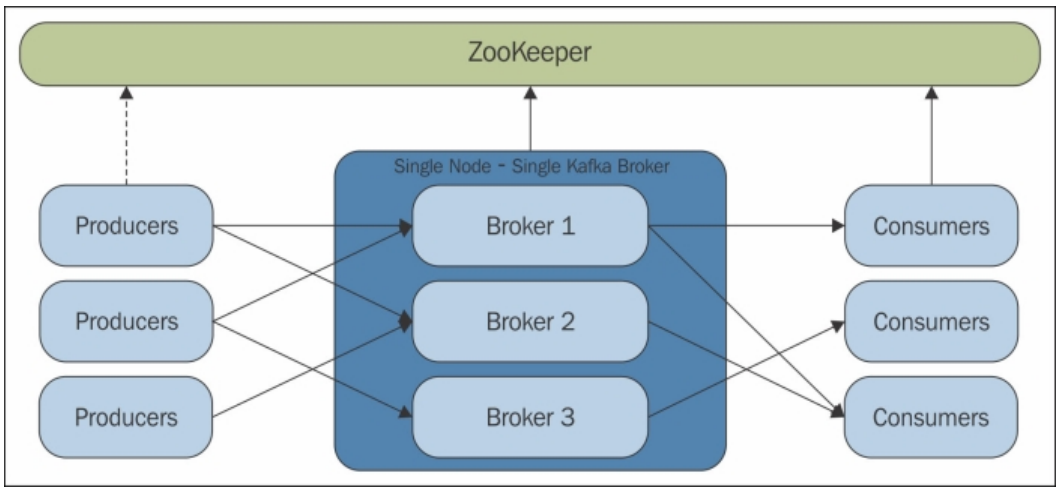

Illustration 11: Kafka Architecture

2.7 ELK Stack

Elastisearch Logstash Kibana is an open source stack of three applications that work together to provide an end-to-end search and visualisation platform for the investigation of log file sources in real time. Log file records can be searched from a variety of log sources, charts, and dashboards built to visualize the log records. The three applications are Elasticsearch, Logstash, and Kibana.

2.7.1 Elasticsearch

Elasticsearch[11] is a distributed, RESTful search and analytics engine capable of solving a growing number of use cases. As the heart of the Elastic Stack, it centrally stores your data so you can discover the expected and uncover the unexpected.

Illustration 12: Elastisearch Logo

2.7.2 Logstash

Logstash[12] is an open source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to your favorite “stash.”

Illustration 13: Logstash Logo

2.7.3 Kibana

Kibana[13] lets you visualize your Elasticsearch data and navigate the Elastic Stack.

Illustration 14: Kibana Logo

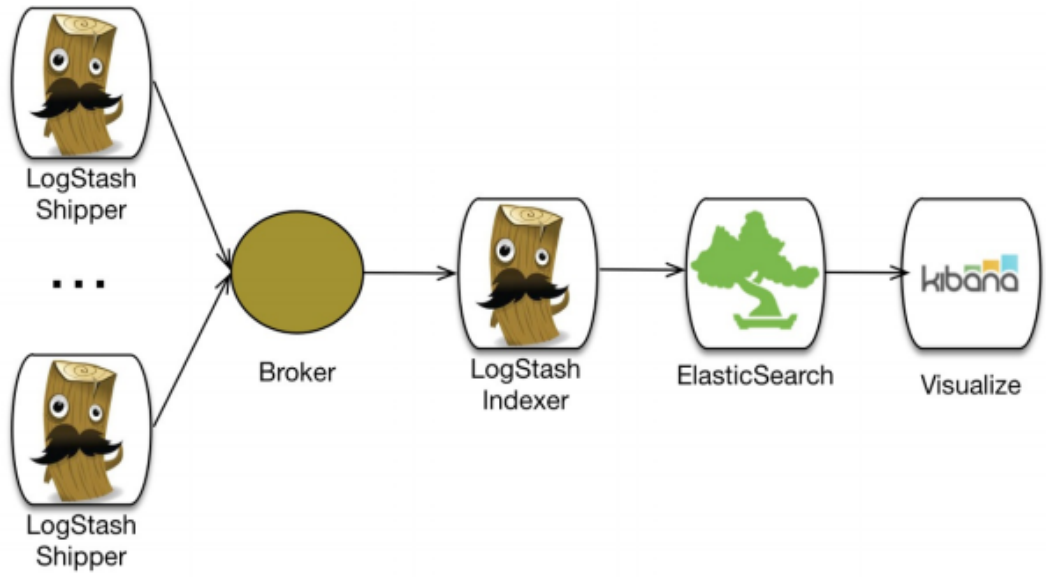

Illustration 15: ELK Stack Diagram

2.8 IBM Bluemix Logging and Monitoring

IBM's platform logging and monitoring solution runs on robust and popular open source offerings today. Logging is delivered from the Elasticsearch ELK stack. Monitoring is delivered from a Graphite and Grafana stack. Both are configured to use a common high-speed Kafka bus.[14]

Illustration 16: IBM Containers Logo

When data is collected from IBM Bluemix Container Service, users benefit from the use of a system crawler. The system crawler automatically collects select compute instance-related logs and metrics. Additionally, the crawler can be configured to collect more, all without agents or sidecar configurations.[14]

Data flows into the logging and monitoring service through an initial Logstash connection point. After the data is in the private network, the data is processed then stored in Elastic Search or Graphite. Users gain access to the data through the Multitenant Proxy.[14]

2.8.1 Monitoring Overview

Container metrics are collected from outside of the container, without having to install and maintain agents inside of the container. In-container agents can have significant overheads and setup times for short-lived, lightweight cloud instances and auto-scaling groups, where containers can be rapidly created and destroyed. This out-of-band data collection approach eliminates these challenges and removes the burden of monitoring from the users.[14]

2.8.2 Logging Overview

Similar to metrics, container logs are monitored and forwarded from outside of the container by using crawlers. The data is sent by the crawlers to a multi-tenant Elasticsearch in Bluemix, just like logs that are collected by other in-container agents, but without the hassle of having to install the agents inside the container.[14]

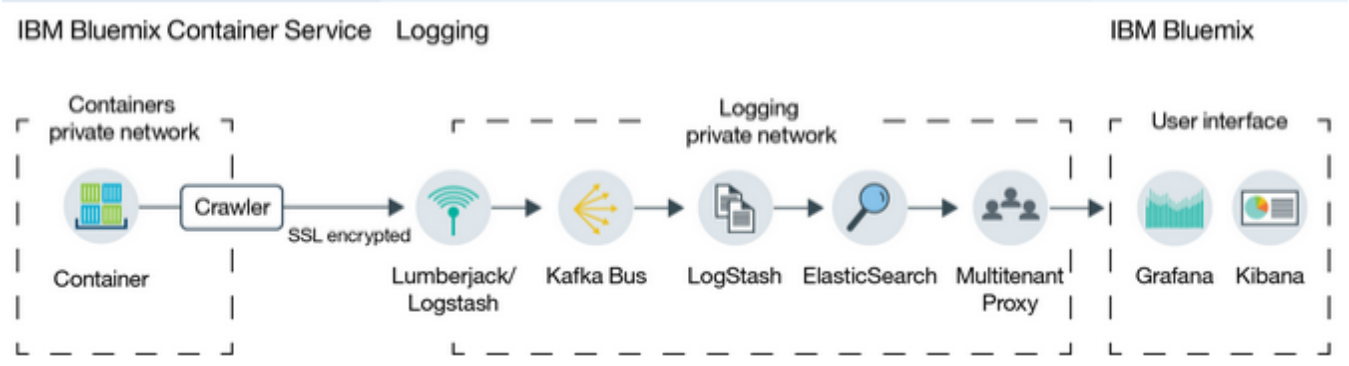

Illustration 17: Bluemix Container Service Logging

2.9 Container Orchestration

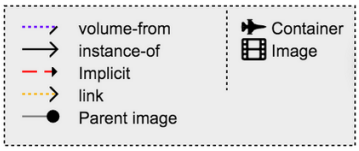

Illustration 18: Container Orchestration Legend

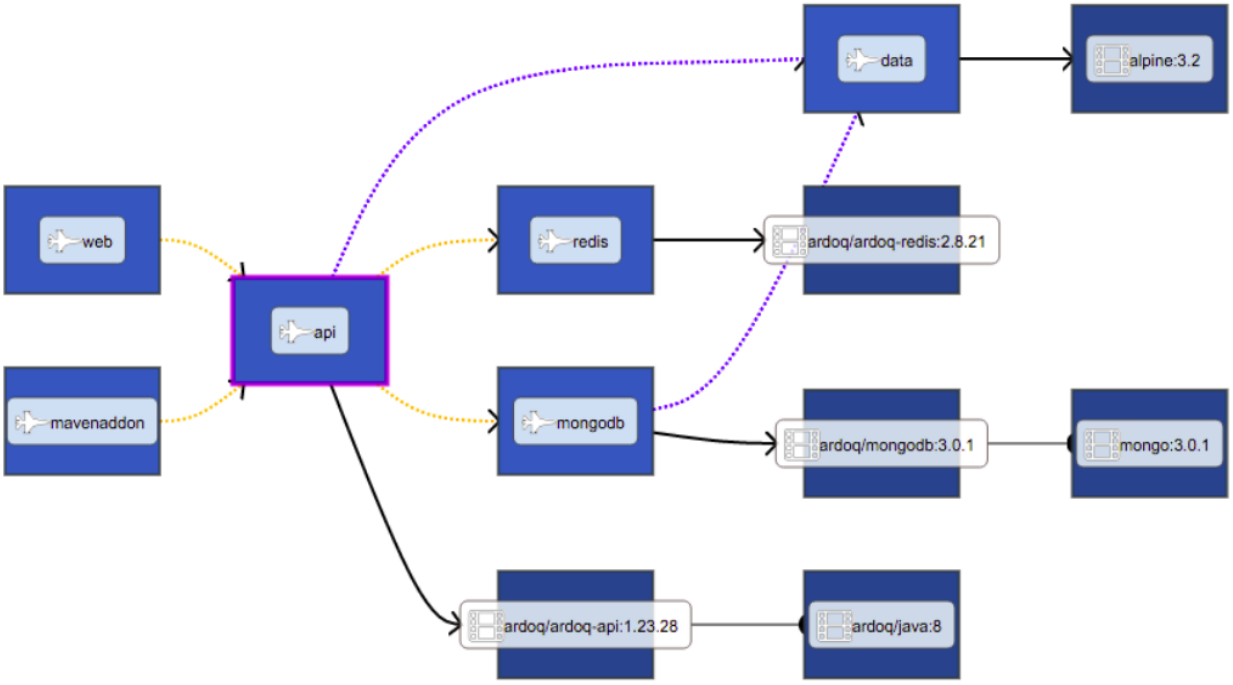

Illustration 19: Container Orchestration

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration.

Compose is great for development, testing, and staging environments, as well as CI workflows.

Using Compose is basically a three-step process.

1. Define your app's environment with a `Dockerfile` so it can be reproduced anywhere.
2. Define the services that make up your app in `docker-compose.yml` so they can be run together in an isolated environment.
3. Lastly, run `docker-compose up` and Compose will start and run your entire app.

The real advantage of Compose is for applications that revolve around a single-purpose server that could easily scale out if architectural complexity is not a requirement. Development environments, which tend to use an all-in-one method of operation, fit well into this requirement.

The community's reception of Compose has been notably positive, but the practicality of its usage and the lack of ability to create a long-term vision around it tend to minimize the actual legitimacy of adopting it as a container orchestration technology.[15]

2.10 Service Discovery

Services typically need to call one another. In a monolithic application, services invoke one another through language-level method or procedure calls. In a traditional distributed system deployment, services run at fixed, well known locations (hosts and ports) and so can easily call one another using HTTP/REST or some RPC mechanism. However, a modern microservice based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically.[16]

2.10.1 Problem

How does the client of a service - the API gateway or another service - discover the location of a service instance?

2.10.2 Forces

- Each instance of a service exposes a remote API such as HTTP/REST, or Thrift etc. at a particular location (host and port)
- The number of services instances and their locations changes dynamically.
- Virtual machines and containers are usually assigned dynamic IP addresses.
- The number of services instances might vary dynamically. For example, an Autoscaling Group adjusts the number of instances based on load.

2.10.3 Solution

When making a request to a service, the client makes a request via a router that runs at a well known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance.

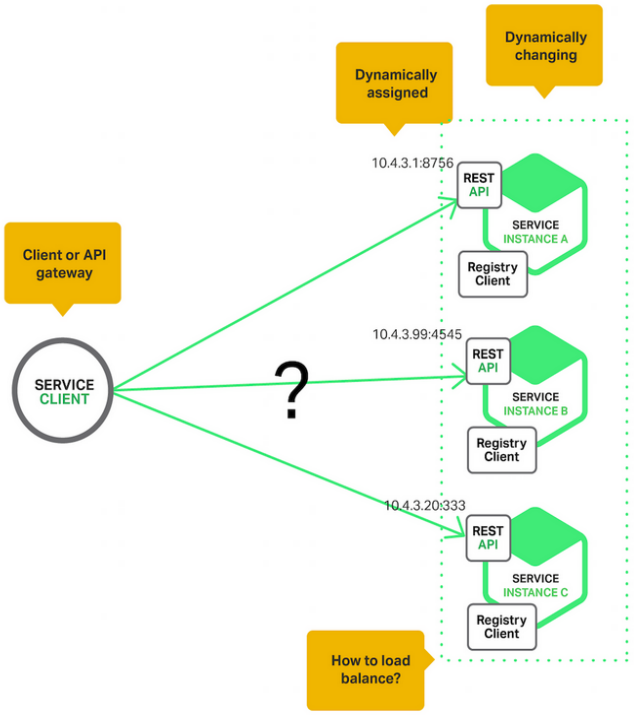

Illustration 20: Service Discovery

2.11 Amalgam8

IBM has made available an open source project named Amalgam8 which takes the tedium out of microservice management, enabling faster development, more control, and better resiliency of microservices without impacting existing implementation code. It provides advanced DevOps capabilities such as systematic resiliency testing, red/black deployment, and canary testing necessary for rapid experimentations and insight.[17]

Illustration 21: Amalgam8 Logo

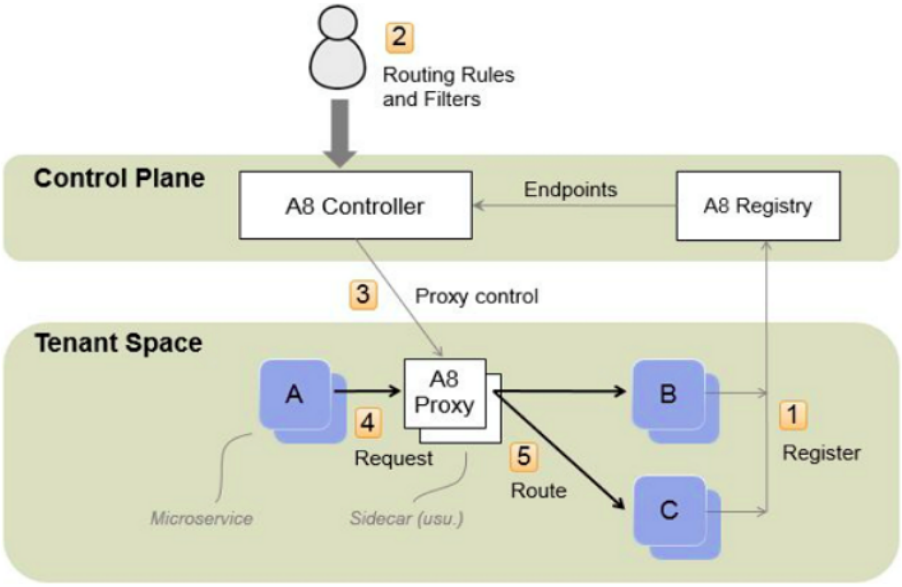

Illustration 22: Amalgam8 Architecture

At the heart of Amalgam8 are two multi-tenanted services:

- ***Registry –*** A high-performance service registry that provides a centralized view of all the microservices in an application, regardless of where they are actually running.
- ***Controller –*** A tool that monitors the Registry and provides a REST API for registering routing and other microservice control-rules, which it uses to generate and send control information to proxy servers running within the application

Applications run as tenants of these two servers. They register their services in the Registry and use the Controller to manage proxies, usually running as sidecars of the microservices.

- Microservice instances are registered in the *A8 Registry*.
- An *Administrator* specifies routing rules and filters (such as version rules and test delays) to control traffic flow between microservices.
- An *A8 Controller* monitors the A8 Registry and administrator input, and then generates control information that is sent to the A8 Proxies.
- Requests to microservices are made through an *A8 Proxy* (usually a client-side sidecar of another microservice).
- The A8 Proxy forwards requests to an appropriate microservice, depending on the request path and headers and the configuration specified by the controller.

2.12 DevOps CI/CD Pipeline

2.12.1 Continuous Integration

Continuous Integration *(CI)* is a strategy for how a developer can integrate code to the mainline continuously - as opposed to frequently.

Illustration 23: IBM Bluemix DevOps Services Logo

CI was created for agile development. It organizes development into functional user stories. These user stories are put into smaller groups of work, sprints. The idea of continuous integration is to find issues quickly, giving each developer feedback on their work and Test Driven Development (TDD) evaluates that work quickly. With TDD, you build the test and then develop functionality until the code passes the test. Each time, when you make new addition to the code, its test can be added to the suite of tests that are run when you build the integrated work. This ensures that new additions don't break the functioning work that came before them, and developers whose code does in fact "break the build" can be notified quickly.[18]

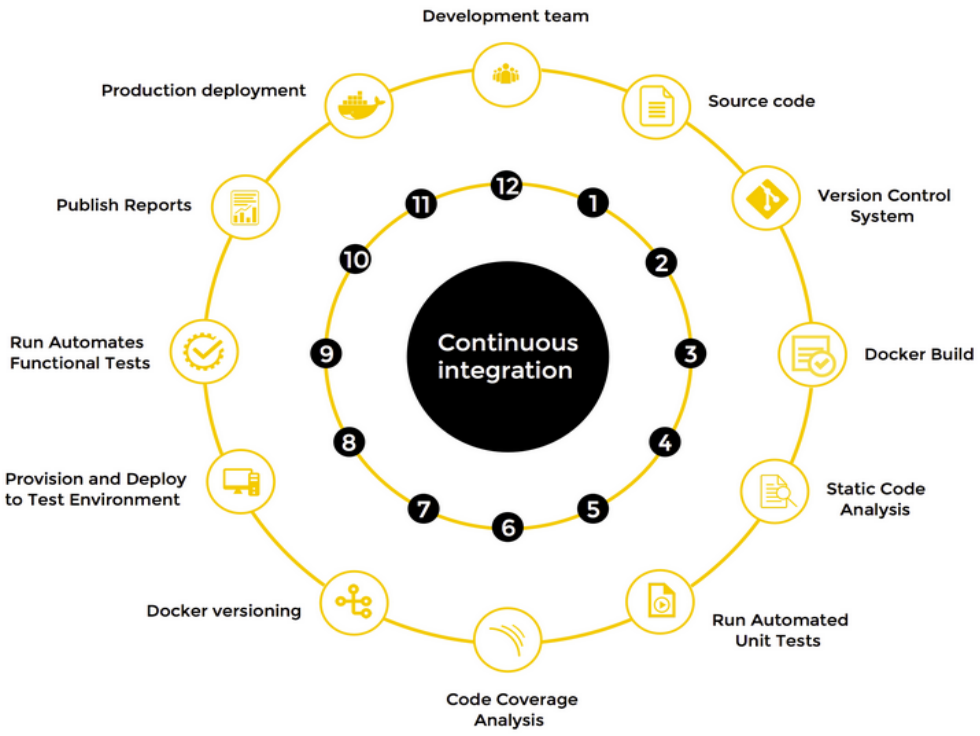

Illustration 24: Continuous Integration

2.12.2 Continuous Delivery

Continuous Delivery (CD) is a core technique that stems from Continuous Integration (CI), where as per business needs, quality software is deployed frequently and predictably to Production in an automated fashion. This improves feedback and reduces shelf-time of new ideas, and thereby improves the sustainability of businesses. Continuous Delivery includes Continuous Deployment and Continuous Testing.

2.13 InfluxDB

InfluxDB is an open-source[26] time series database developed by InfluxData. It is written in Go and optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics.[28]

Illustration 25: InfluxDB Logo

InfluxDB has no external dependencies[27] and provides an SQL-like language with builtin time-centric functions for querying a data structure composed of measurements, series, and points. Each point consists of several key-value pairs called the fieldset and a timestamp. When grouped together by a set of key-value pairs called the tagset, these define a series. Finally, series are grouped together by a string identifier to form a measurement.

Values can be 64-bit integers, 64-bit floating points, strings, and booleans.

Points are indexed by their time and tagset.

Retention policies are defined on a measurement and control how data is downsampled and deleted.

Continuous Queries run periodically, storing results in a target measurement.[29]

Continuous queries are created when you issue a select statement with an `into` clause. Instead of returning the results immediately like a normal select query, InfluxDB will instead store this continuous query and run it periodically as data is collected. Only cluster and database admins are allowed to create continuous queries.[29]

Continuous queries let you precompute expensive queries into another time series in real-time.

Example Query:

```
select count(schuman) from events

group by time(5m), type

into events.ts.5m
```

When a continuous query is created from a select query that contains a group by time() clause, InfluxDB will write the aggregate into the target time series when each time interval elapses. On creation, the cluster will backfill old data asynchronously in the background.

```
select count(schuman) from clicks group by time(1h) into clicks.count.1h
```

Each hour, this query will count the number of points written into the time series called `clicks` and write a single point into the target time series called `clicks.ts.1h`. It's important to note that this happens as soon as the hour has elapsed.[29]

2.14 REST and MQTT

2.14.1 REST and MQTT Mirroring

This technique is applied to all REST API services that manage resources that can change state.**[31]**

REST poses a problem when services depend on being up to date with data they don't own and manage. Being up to date requires polling, which quickly add up in a system with enough interconnected services. Microservices need to know when the state they are interested in changes without unnecessary polling.[31]

The mirroring technique uses the end-point URL as an MQTT topic. A downstream microservice using a REST API endpoint can also subscribe to the MQTT topic that matches the end-point syntax. A microservice starts by making an HTTP GET request to obtain the initial state of the resource. Subsequent changes to the resource made by any other microservice are tracked through the MQTT events.[31]

1. The service with a REST API will field requests by client services as expected. This establishes the baseline state.[32]
2. All the services will simultaneously connect to a message broker.
3. API service will fire messages notifying about data changes (essentially for all the verbs that can cause the change, in most cases POST, PUT, PATCH and DELETE).
4. Clients interested in receiving data updates will react to these changes according to their functionality.
5. In cases where having the correct data is critical, client services will forgo the built-up baseline + changes state and make a new REST call to establish a new baseline before counting on it.

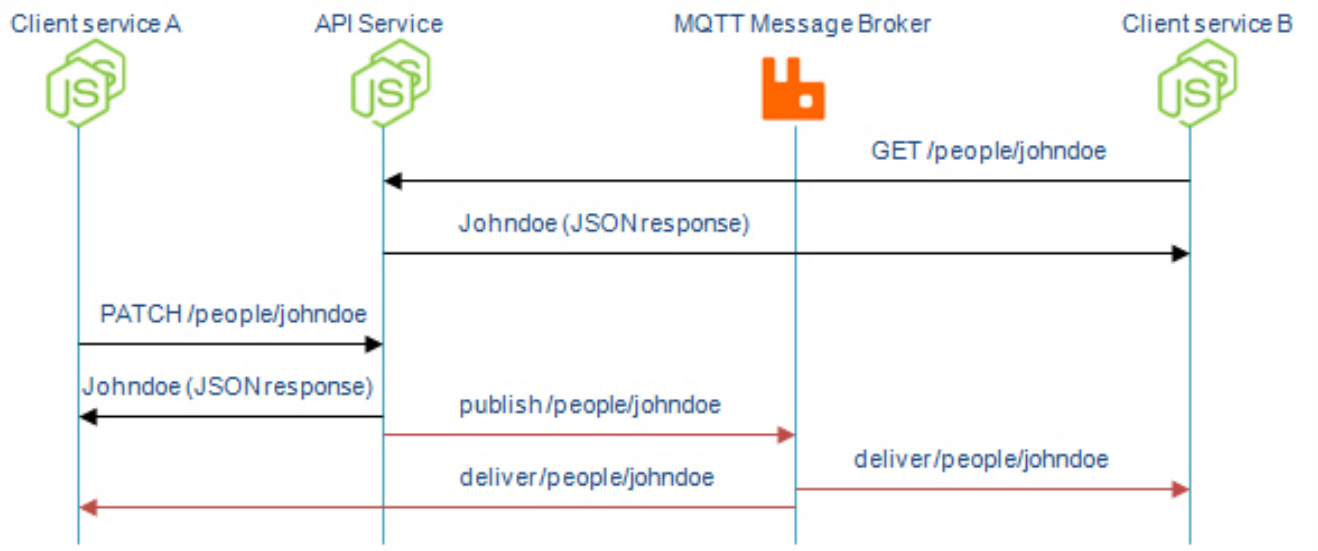

Illustration 26: REST and MQTT Mirroring

2.14.2 Separate Interfaces

Another set of abstractions is to enable the configuration and of existing message systems such as MQTT using REST endpoints. There are two interfaces, a message interface and a REST interface, to shared stored resources. There are two message operations supported, to enable clients or brokers to publish updates to REST endpoints through a message interface, and to publish updates of REST endpoints to brokers or clients subscribing through a message interface. There is also a subscribe operation using the message interface. Incoming messages update resources, and resource updates send outgoing messages.[33]

The convention is to use URIs as topics that allow the system to construct URLs for the REST resource endpoints. The payload consists of the resource representation, e.g. JSON. This handles incoming and outgoing publish operations and acts as a message broker with REST access to topics and, if there is storage, topic history storage. REST Message Broker allows message clients to publish to REST endpoints and subscribe to REST endpoints. An application can monitor an MQTT endpoint by observing the REST resource it publishes to, and an application can publish to an MQTT client by updating to the REST endpoint the MQTT endpoint is subscribed to. For example, sensors could connect with MQTT and publish updates for application software to read, and application software could update a REST endpoint, resulting in the system publishing the update to a sensor that is subscribed to the URL as topic.[33]

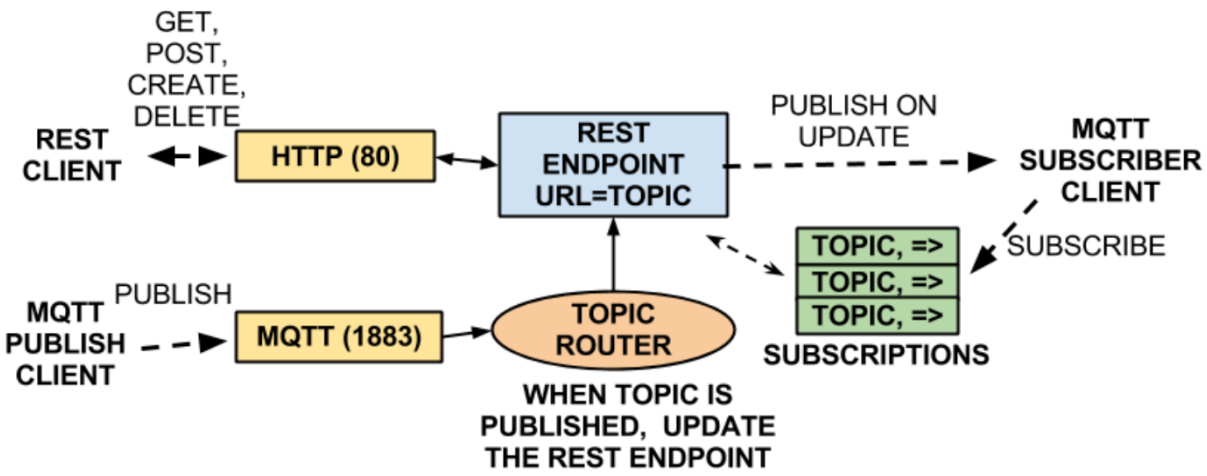

Illustration 27: REST and MQTT Interface

The above example shows how an MQTT broker endpoint can be added to a REST server to enable REST clients and MQTT clients to share data by mapping URLs to topics. When either a REST client updates a resource an MQTT client publishes to a topic, the corresponding resource update is published to any subscribers.

2.15 Similar Projects

2.15.1 TRANSIT: Flexible pipeline for IoT data with Bluemix and OpenWhisk

This Blog post[19] was the first tutorial found that helped explain how to interface Node-Red with OpenWhisk. The proposed architecture uses the Watson IoT Platform and Message Hub in order to handle MQTT client messages. These messages are published to a Kafka server which in turns uses OpenWhisk to decode the base64 encoded data. The data is stored using IBM Object Storage which is then sent to Apache Spark for analytics.

This article helped with understanding IoT simulation using Node-Red, however the architecture of this theses is aimed more towards using Containers to handle the load balancing of MQTT clients.

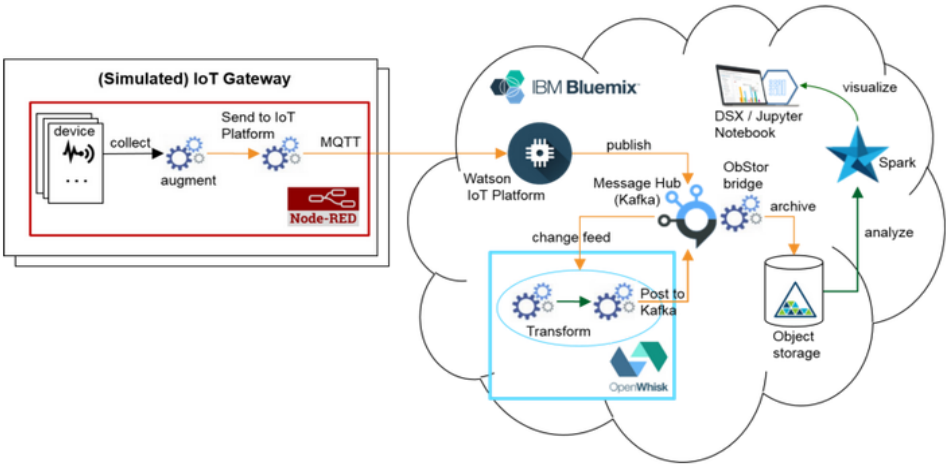

Illustration 28: TRANSIT Diagram

2.15.2 How to Build a High Availability MQTT Cluster for the Internet of Things

A Significant amount of the proposed architecture for this project is borrowed from this blog post.[20] It details the following.

- Setting up a Mosca Broker with Redis
- Dockerizing the MQTT server
- Adding HAProxy as a load balancer
- Access Control List configuration
- Making MQTT secure with SSL

A more detailed summary of the problems encountered with this implementation will be discussed in section 4.

3 Architecture

3.1 Autumn High Level Design Diagram

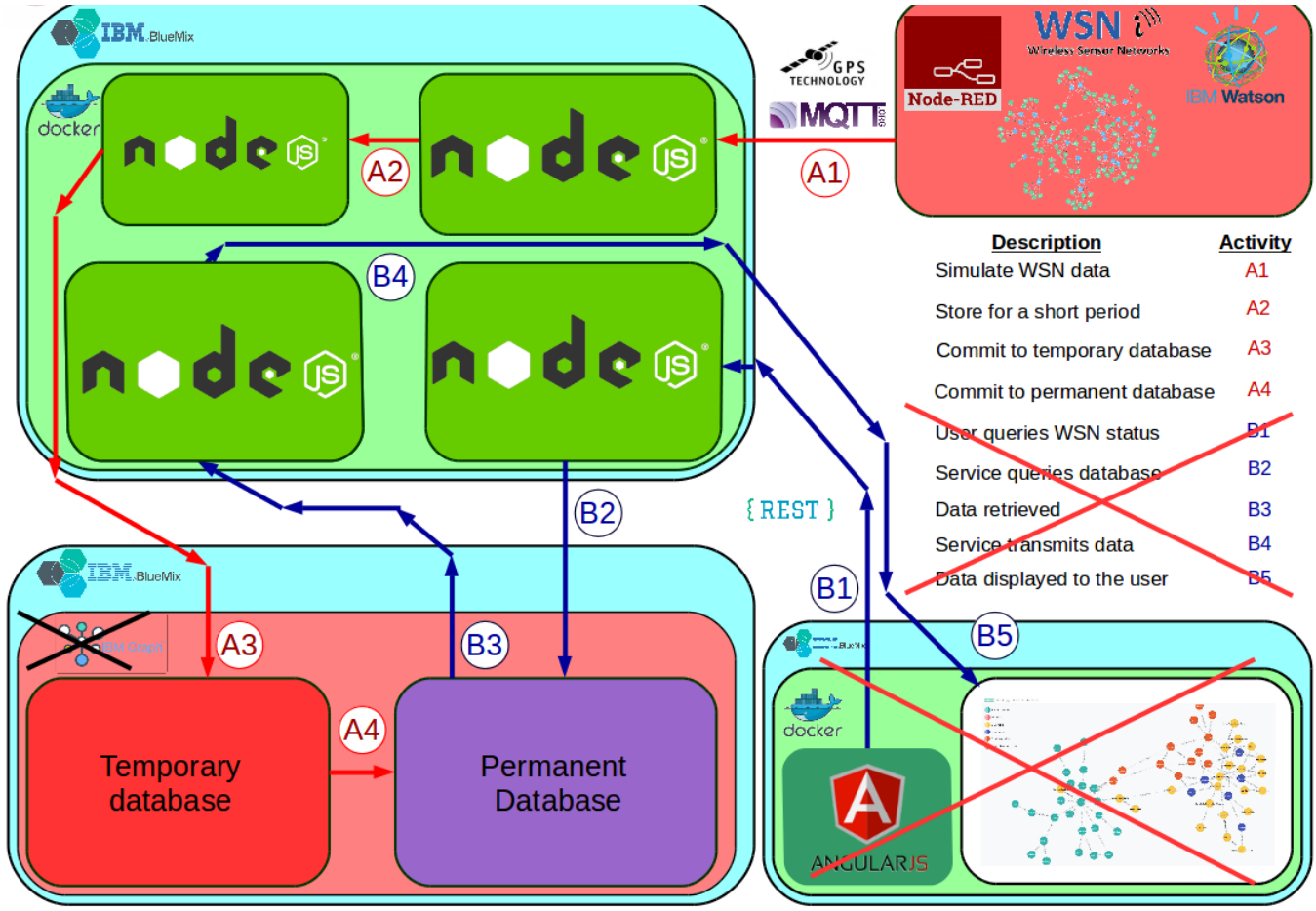

Illustration 29: Autumn High Level Design

The above figure shows the high level design for the theses at the end of the Autumn semester. At the advise of the supervisor, the front-end Angular application was removed from the scope of the project. This was done to due to time constraints.

The other major change in the project is the removal of the Graph database. This was done for two reasons. The first is the time constraint to both learn and implement Graph databases. Secondly Cloudant and InfluxDB cover the use cases for the system, specifically geospatial and time-centric queries respectively.

The high level diagram simply detailed the major functional requirements but lacked a deeper understanding of the problems faced with developing the project.

3.2 Overall Architecture

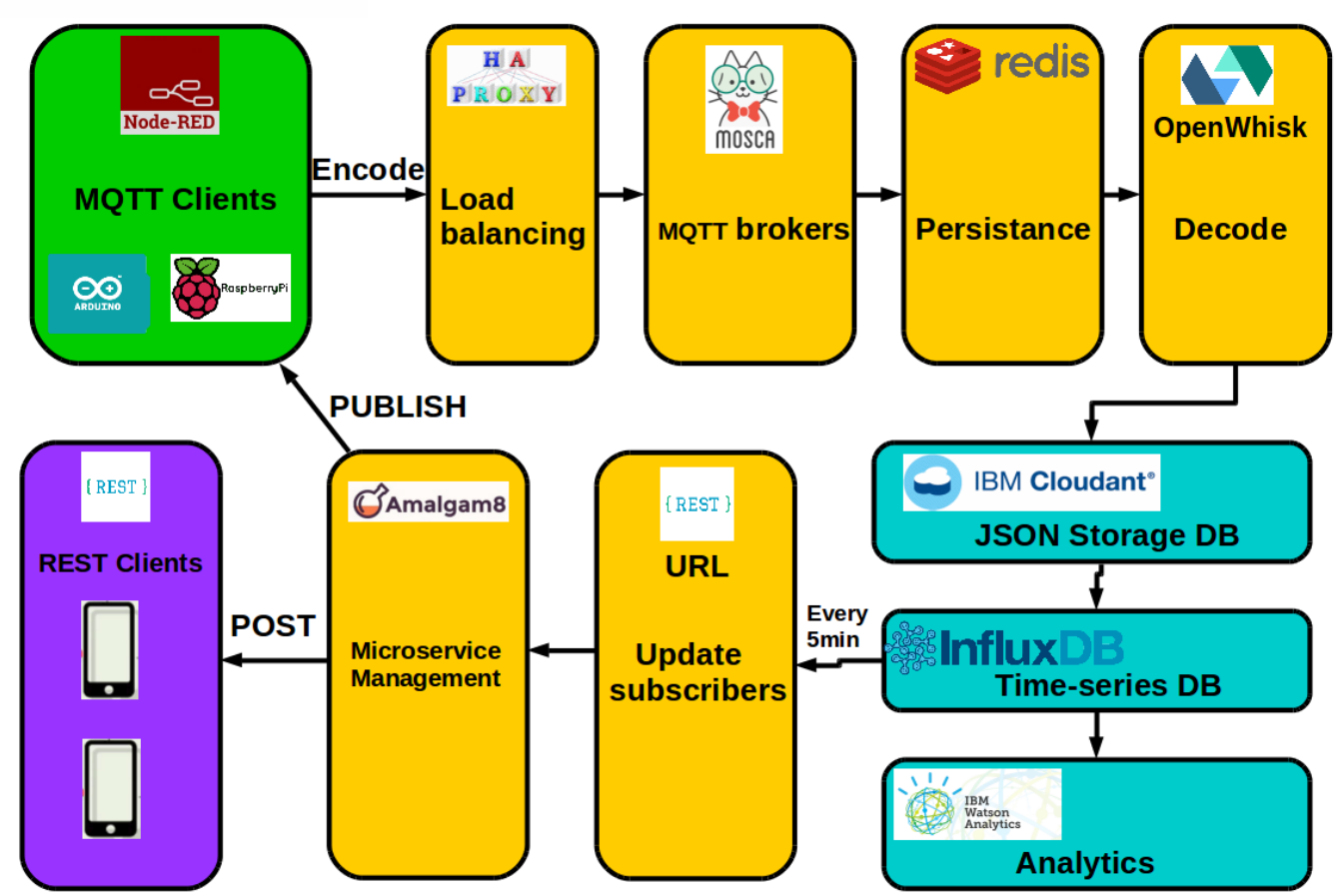

Illustration 30: Overall Architecture

Node-RED simulates MQTT clients and injects GeoJSON messages encoded in base64 to the HAProxy Container. These messages are then load balanced between two Mosca brokers that cache the data in Redis for persistence. A serverless OpenWhisk action is triggered to decode the data from base64 and store it in Cloudant as JSON.

InfluxDB will organise the data into a time-series for efficiency. The continuous queries will be performed in relation to both vital information for microservices registered in the A8 registry as well as various Watson Analytics applications. The continuous queries precompute expensive queries into another time series in real-time.

The topic URL has MQTT clients subscribed to it. Both MQTT and REST clients will be updated once the Amalgam8 load balancer decides which version of the microservice to run. These services can be viewed as downstream services, their execution is dependent on the InfluxDB queries which are acting as upstream services for both Amalgam8 and the IBM Watson Analytics platform.

3.3 Load Balancing Architecture

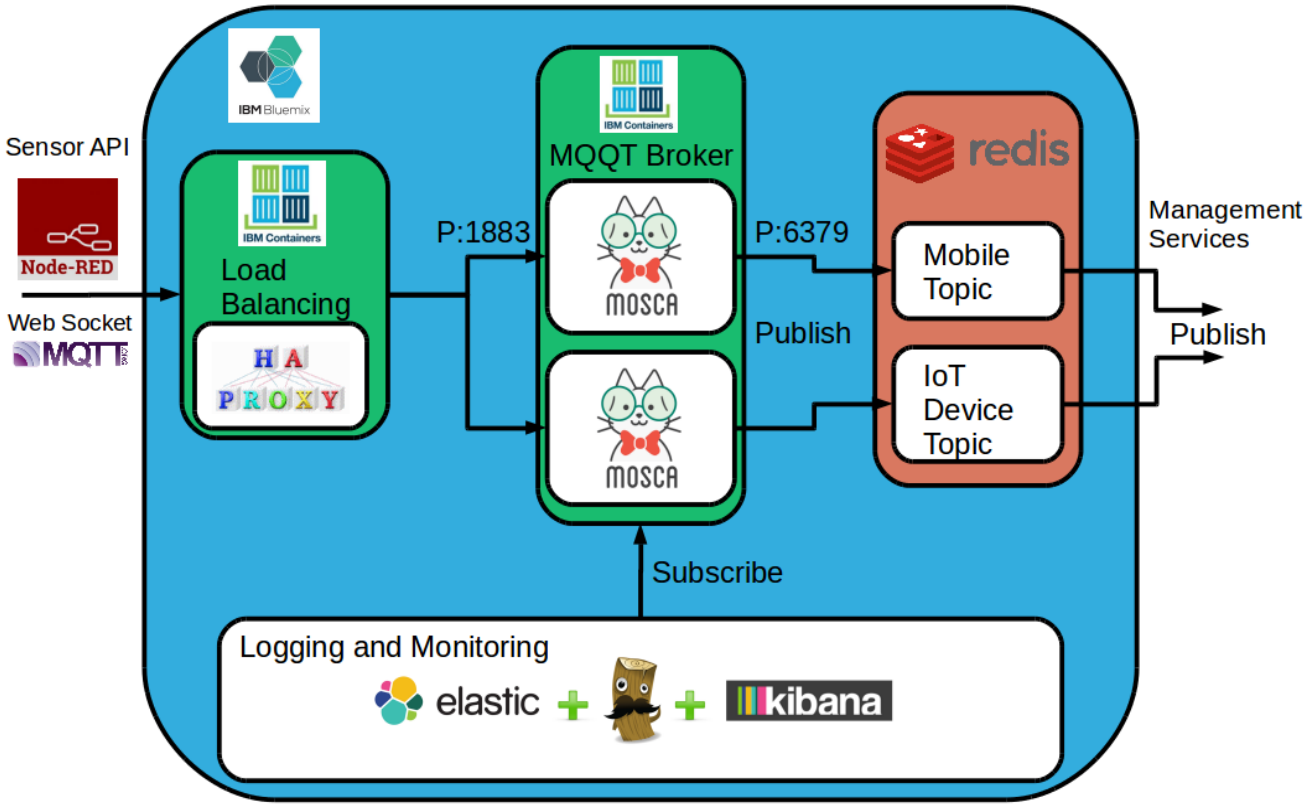

Illustration 31: Loadbalancing Architecture

The scope of this report focuses more on the work completed in relation to the above figure. Sections 4.2-4.5 describe in more detail how both the HAProxy and Mosca Containers are configured and deployed.

SSL traffic comes from the IoT gateway. It is then decrypted and load balanced between two Mosca brokers that use Redis in order to deliver messages.

In past monolithic architecture if there was a failure, the administrator would simply log into the machine in order to view the logs. This is not possible in a microservices architecture, extensive external logging and monitoring is required. The ELK stack will be used to log and monitor the health of the Mosca brokers.

Redis will publish to the management services that decide what to do with the given data. The problem of distinguishing between a sensor sending data and a sensor calling a command has not yet been sufficiently reviewed. It is assumed for now that sensors will only be publishing data to the car-iotp and mobile-iotp topics.

4 Theory

4.1 Mosca Node.js Code

Illustration 32: Mosca Logo

4.1.1 Mosca-Redis Configuration

```
var mosca = require('mosca')

var ascoltatore = {
  type: 'redis',
  redis: require('redis'),
  db: 12,
  port: 6379,
  return_buffers: true, // to handle binary payloads
  host: "localhost"
};

var moscaSettings = {
  port: 1883,
  backend: ascoltatore,
  persistence: {
    factory: mosca.persistence.Redis
  }
};
```

Illustration 33: Mosca Redis COnfiguration

Ascoltatore focuses on providing a simple and unique abstraction for all supported brokers, in this case Redis. The moscaSettings variable is configured to connect to the Redis server using Ascoltatore.

```
persistence: {
  factory: mosca.persistence.Redis,
  host: 'your redis host',
  port: 'your redis port'
}
```

Illustration 34: Redis Persistence

If using a non-standard redis port or redis is running on a different server, it is necessary to set the host and port in the persistence section. Running Redis as a container or using the Redis service on Bluemix is still not decided upon at this point in time.

4.1.2 Authentication and Authorization

With Mosca it is possible to authorize a client defining three methods.

- **`authenticate`**
- **`authorizePublish`**
- **`authorizeSubscribe`**

These methods can be used to restrict the accessible topics for specific clients. Below is an example of a client that sends a username and a password during the connection phase and where the username will be saved and used later on. (To verify if a specific client can publish or subscribe for the specific user).[35]

```
var authenticate = function(client, username, password, callback) {
  var authorized = (username === 'alice' && password.toString() === 'secret')
  if (authorized) client.user = username;
  callback(null, authorized);
}
```

Illustration 35: Mosca Authenticate

The function “authenticate” accepts the connection if the username and password are valid.

```
var authorizePublish = function(client, topic, payload, callback)
  callback(null, client.user == topic.split('/')[1]);
}
```

Illustration 36: Mosca Authorize Publish

In this case the client authorized as alice can publish to /users/alice taking the username from the topic and verifying it is the same as the authorized user.

```
var authorizeSubscribe = function(client, topic, callback)
  callback(null, client.user == topic.split('/')[1]);
}
```

Illustration 37: Mosca Authorize Subscribe

In this case the client authorized as alice can subscribe to /users/alice taking the username from the topic and verifying it is the same of the authorized user.

With this logic someone that is authorized as 'alice' will not be able to publish to the topic `users/bob`.

This logic will be used to prevent car IoT devices publishing/subscribing to the “mobile-iotp” topic and similarly prevent mobile phone devices publishing/subscribing to the “car-iotp” topic.

It is important to note that the code described in this section has not yet been implemented into the project.

4.2 Bluemix Container Service

4.2.1 IBM Container Service Login

Illustration 38: Bluemix Logo

```
tom@tom-pc:~/bluemixcli/Bluemix_CLI$ cf login -a api.eu-gb.bluemix.net
API endpoint: api.eu-gb.bluemix.net

Email> 16117743@studentmail.ul.ie

Password>
```

Illustration 39: Container Service Login

4.2.2 Setting Namespace and Authentication with IBM Container's Registry

```
tom@tom-pc:~/BM/mosca/mosca$ cf ic namespace set tomspace2
tomspace2
tom@tom-pc:~/BM/mosca/mosca$ cf ic init
Deleting old configuration file...
Generating client certificates for IBM Containers...
Storing client certificates in /home/tom/.ice/certs/...

Storing client certificates in /home/tom/.ice/certs/containers-api.ng.bluemix.net/f3

OK
The client certificates were retrieved.

Checking local Docker configuration...
OK

Authenticating with the IBM Containers registry host registry.ng.bluemix.net...
OK
You are authenticated with the IBM Containers registry.
Your organization's private Bluemix registry: registry.ng.bluemix.net/tomspace2
```

Illustration 40: Setting Namespace

4.2.3 Bluemix Dashboard

Illustration 41: Bluemix Dashboard

4.2.4 Cloning Mosca Repository From Github

```
tom@tom-pc:~/BM/mosca$ git clone https://github.com/mcollina/mosca.git
Cloning into 'mosca'...
remote: Counting objects: 4681, done.
```

Illustration 42: Cloning Mosca Repository

4.2.5 Pushing Mosca Image to the "tomspace2" Registry

```
tom@tom-pc:~/BM/mosca/mosca$ cf ic build -t mosca-image .
Sending build context to Docker daemon 7.669 MB
Step 1 : FROM mhart/alpine-node:4
4: Pulling from mhart/alpine-node
```

Illustration 43: Pushing Mosca Image

4.3 Mosca Broker Container

The Mosca image was pushed to the Bluemix image registry after testing the functionality of the Mosca Broker on the local Ubuntu machine. The test consisted of using the Mosquitto service installed on the local machine. Mosquitto verified the Mosca Broker's functionality by publishing and subscribing in different terminals.

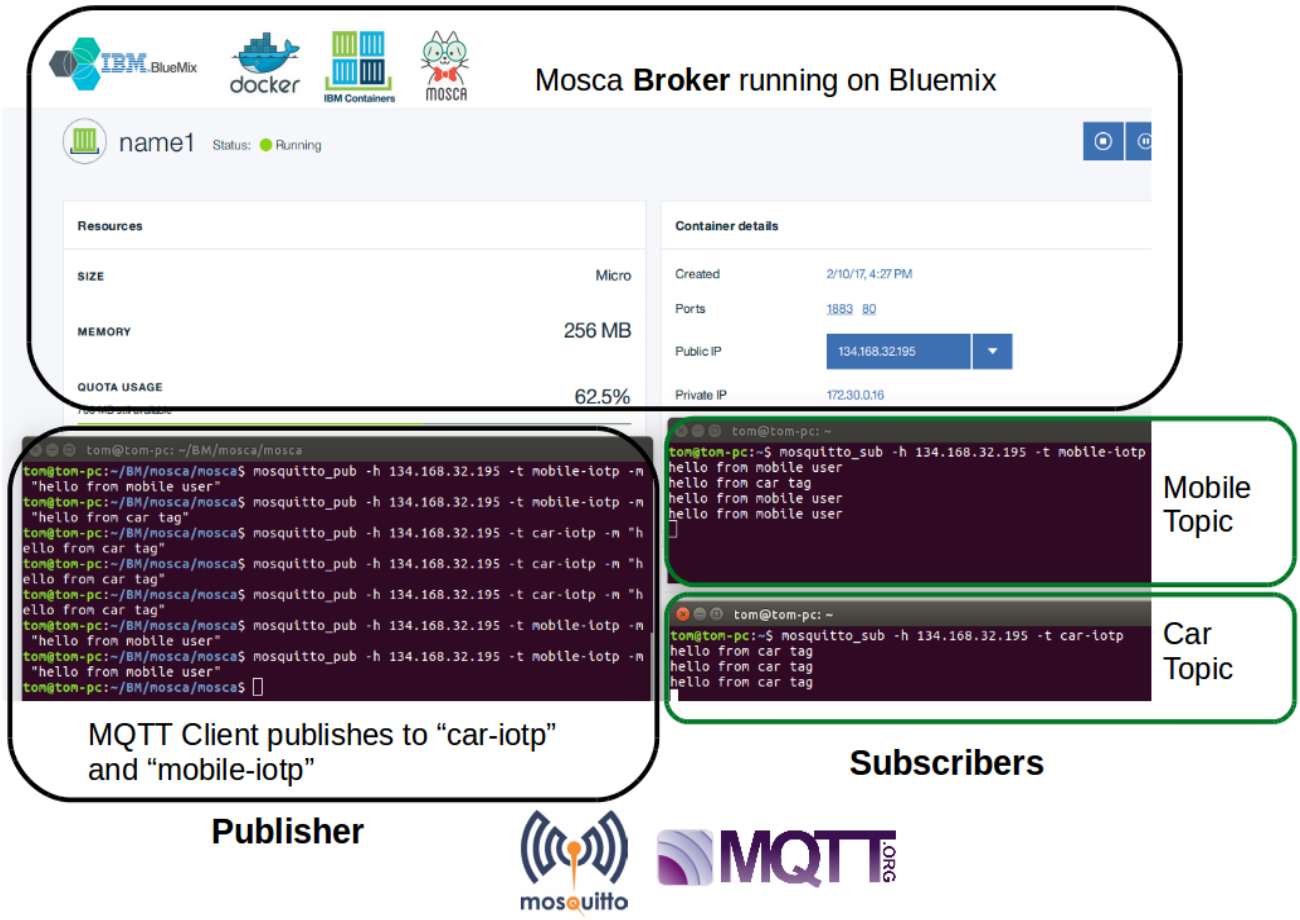

Illustration 44: Mosca Broker Test

4.3.1 Broker

The top half of the above figure shows the Bluemix Dashboard. It displays the Mosca Broker running in a Container called "name1". It has the ports 80 and 1883 exposed.

4.3.2 Mosquitto

The terminal on the left is using Mosquito to publish to the topics "mobile-iotp" and "car-iotp". These messages are being sent to the Mosca Broker Container running on Bluemix.

The terminals on the right are using Mosquito to subscribe to the mobile-iotp and car-iotp topics.

4.4 Mosca Broker Container Group

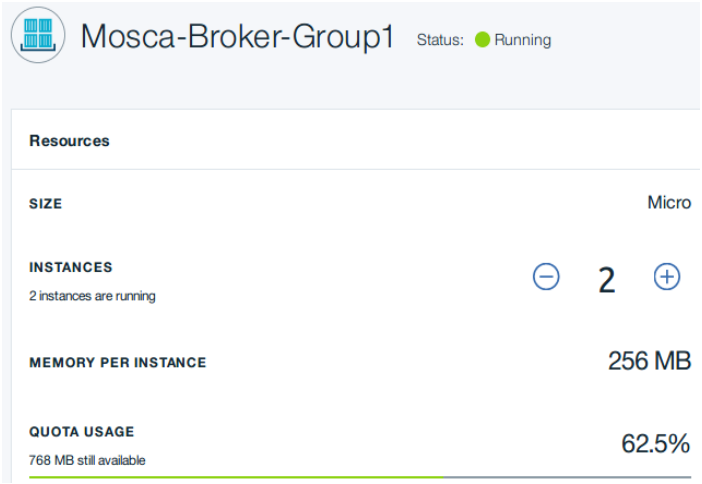

Illustration 45: Container Group

An IBM Container Scalable Group consists of multiple containers that all share the same image. The original architecture for this project heavily depended on using Container Groups for scaling out MQQT brokers. It turns out that IBM Containers Scalable Groups do not support non-HTTP traffic for the exposed ports. In a Scalable Group, an external URL is bound to the Go router serving the platform. HTTPS requests made to port 80 on the external URL are sent to the port specified during configuration on the internal container hosts. Direct external access to the ports on the external container is not allowed.

Further research into a solution to this problem is required. If Scalable Groups cannot be implemented for the Mosca Brokers then a manual Container Orchestration solution will have to be drawn up. This would be configured using the docker-compose CLI.

```
docker-compose scale broker=3
```

The problems that need to be solved with this approach.

- Configuration of detecting “scale out” “scale in” triggers
- Configuring Docker Remote API
- Each Mosca Broker will need to be configured with the required ssl certs to make a call using the Docker Remote API in order to scale out or in
- Figuring out how the Docker Remote API is integrated with Bluemix

4.5 HaProxy Container Configuration

Below is a HAProxy config file for load balancing between 2 Mosca Brokers. The HAProxy listens for all requests coming to port 1883 and forwards them to the MQTT servers (mosca_1 and mosca_2) using the *leastconn* balance mode (selects the server with the least number of connections).

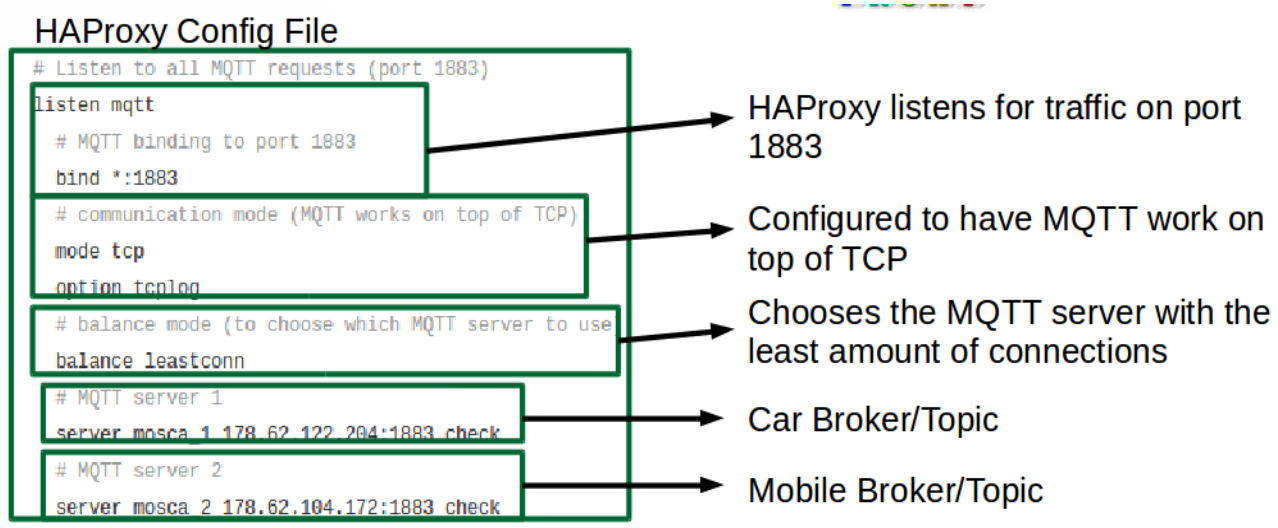

Illustration 46: HAProxy Config

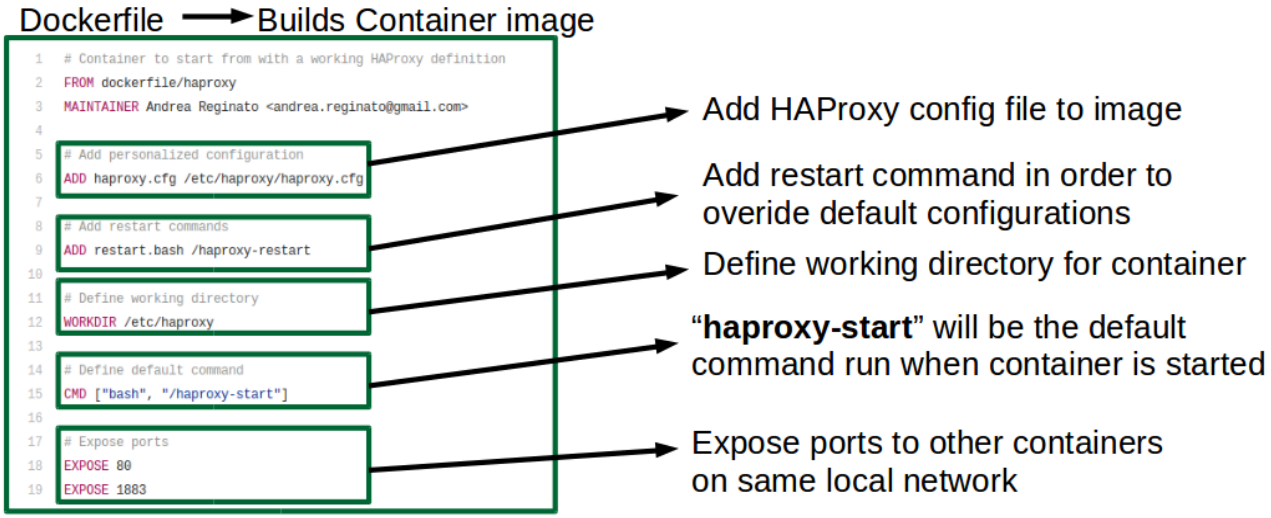

Illustration 47: HAProxy Dockerfile

The above figure shows the configuration for the dockerfile in order to build the haproxy image.

4.5.1 Problems Encountered

A common problem found with tutorials in relation to Docker is that the examples are quickly outdated. For example

FROM dockerfile/haproxy

needs to be changed to

FROM library/haproxy

After building a HAProxy container with the above Dockerfile. The command "**docker logs <haproxy container id>**" is run in order to determine if the build was successful. An error was reported in relation to the command

CMD ["bash", "/haproxy-start"]

The error states that the "haproxy-start" cannot be found. A quick fix to remove this error was to change the line to

CMD ["bash", "/haproxy-restart"]

This was done merely in a problem solving effort to remove the error to get the container to build.

Another problem encountered was attaching the "overide-config" file as a volume to the container. The process is as follows.

Build and run the container using the original haproxy image with the Dockerfile defined above.

docker pull library/haproxy

docker run -d -p 80:80 library/haproxy

Then issue the following command to the container once it is running.

docker run -d -p 1883:1883 -v <override-dir>:/haproxy-override dockerfile/haproxy

The HAProxy container accepts a configuration file as data volume option (-v), where <*override-dir*> is an absolute path of a directory that contains *haproxy.cfg*.

Other problems such as "cannot find user id for 'haproxy' " had to be solved as well in order to remove errors when debugging with the "**docker logs <haproxy container id>**" command.

4.6 Node-Red

4.6.1 Primary Flow

This flow represents the the core functionality of the theses. Data is flowing from the simulated IoT gateway to the Mosca broker hosted on Bluemix. The data is then decoded using OpenWhisk and stored as JSON in a Cloudant database.

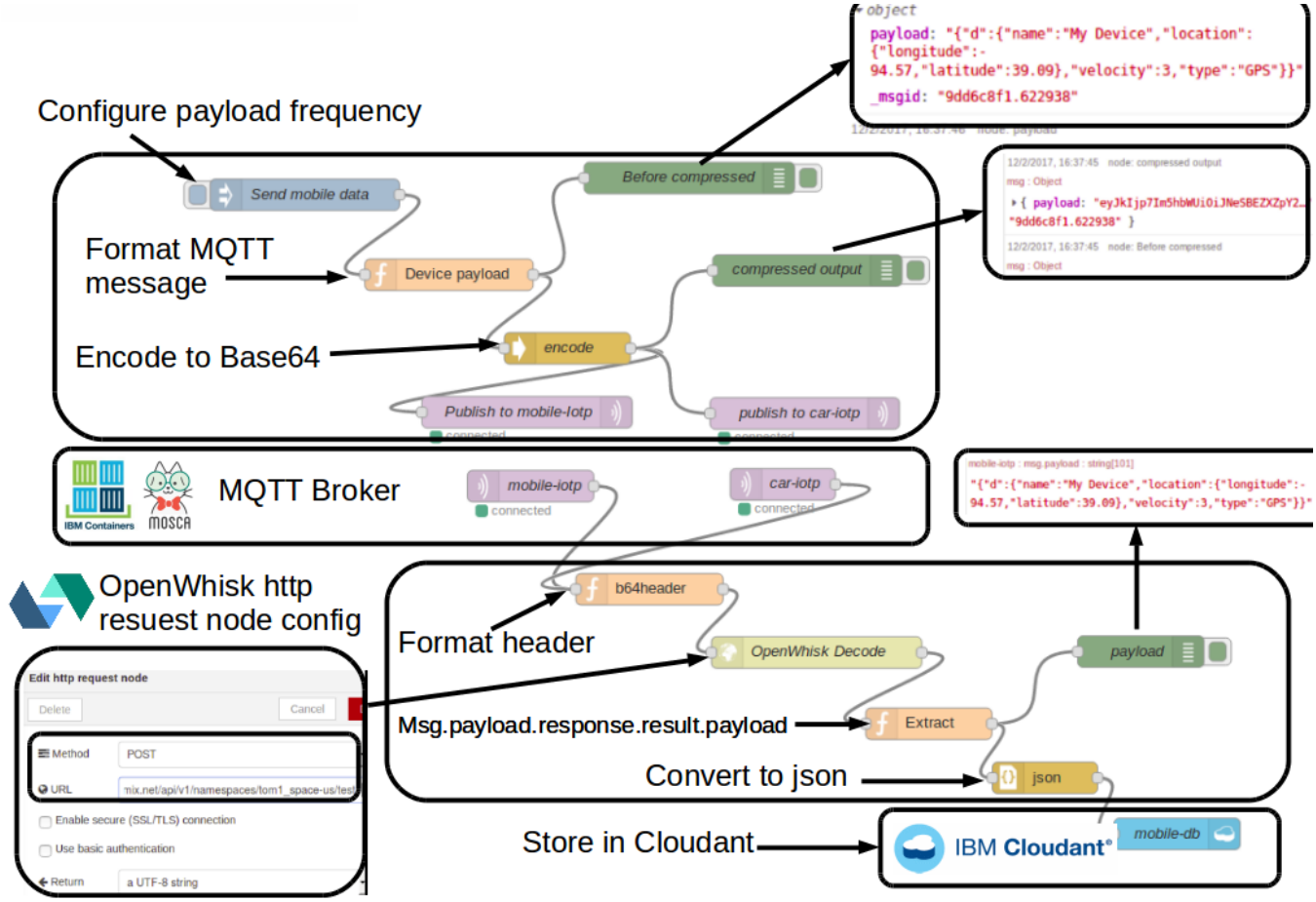

Illustration 48: Primary Flow

4.6.2 Test Cloudant Flow

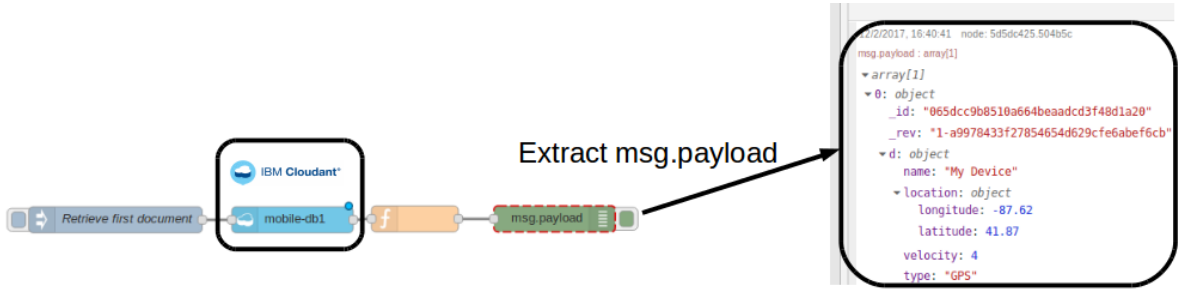

Illustration 49: Test Cloudant Flow

4.7 OpenWhisk

4.7.1 Base64 Decode Action Configuration

Illustration 50: OpenWhisk Decode Action

4.7.2 Node-RED http post configuration

Node-RED has an OpenWhisk node available that makes it straightforward to call actions, however invoking the action through the http request node was done as a quick fix for the problems encountered using the OpenWhisk node.

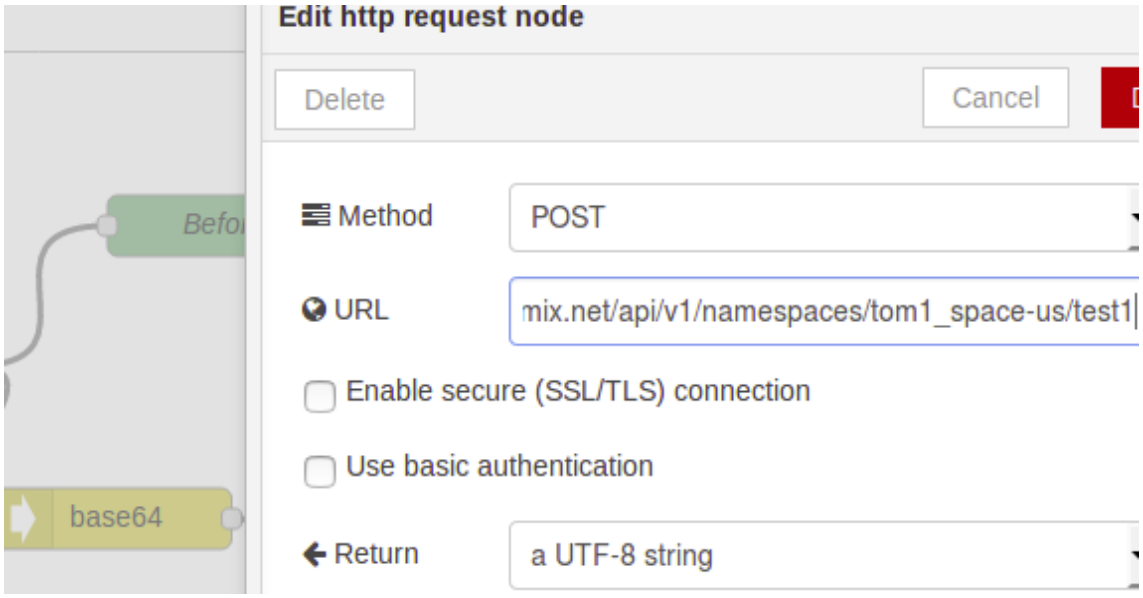

Illustration 51: Http Request Node

4.8 Continuous Deployment Pipeline

4.8.1 Deployment Fails at Validation Stage

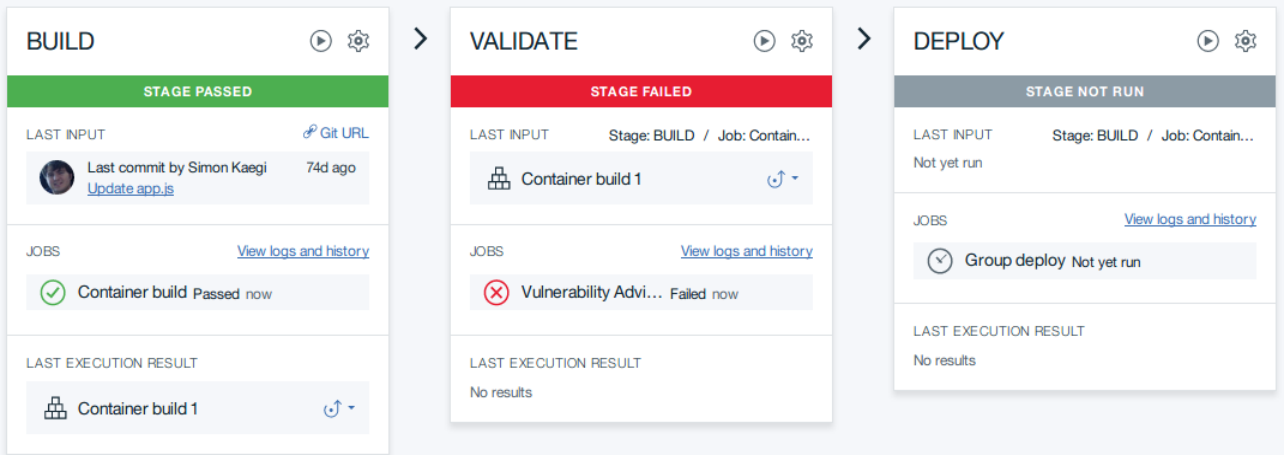

Illustration 52: Deploy Fails

4.8.2 Package Vulnerability

```
image registry.ng.bluemix.net/tomspace2/hello-containers-secure-container-tool
       376 packages scanned
       1 vulnerable packages
              openssl : current: 1.0.1t-1+deb8u5  fixed: 1.0.1t-1+deb8u6
**********************************************************************
2017-02-11 16:56:20,426 - pipeline - INFO - For a more in-depth review of thes
a256:de0f7fe258e969ad3665d52a8e1731739bd1bff9d6ea57d0737ad30976b66225&spaceGui
Script completed in 34.4133069515 seconds
2017-02-11 16:56:20 UTC : Archive directory is /tmp/extension_archive - copyin
2017-02-11 16:56:20 UTC : Issues found in or while getting compliance results.
To send notifications, set SLACK_WEBHOOK_PATH or HIP_CHAT_TOKEN in the environ
2017-02-11 16:56:20 UTC : Script runtime of 0m 36s

Finished: FAILED
```

Illustration 53: Package Vulnerability

4.8.3 Vulnerability Advisor Dashboard

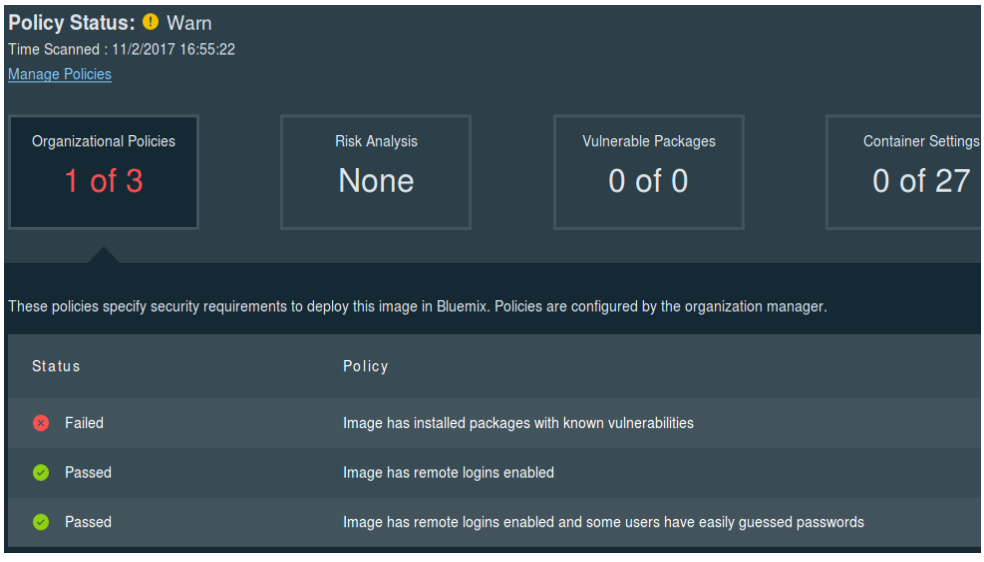

Illustration 54: Vulnerability Advisor Dashboard

4.8.4 Validation Stage Configuration

In order to complete the tutorial example, the validation stage was configured to continue to the deployment stage regardless of vulnerabilities discovered.

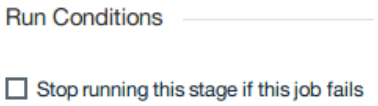

Illustration 55: Edit Run Conditions

4.8.5 Validation Stage Running

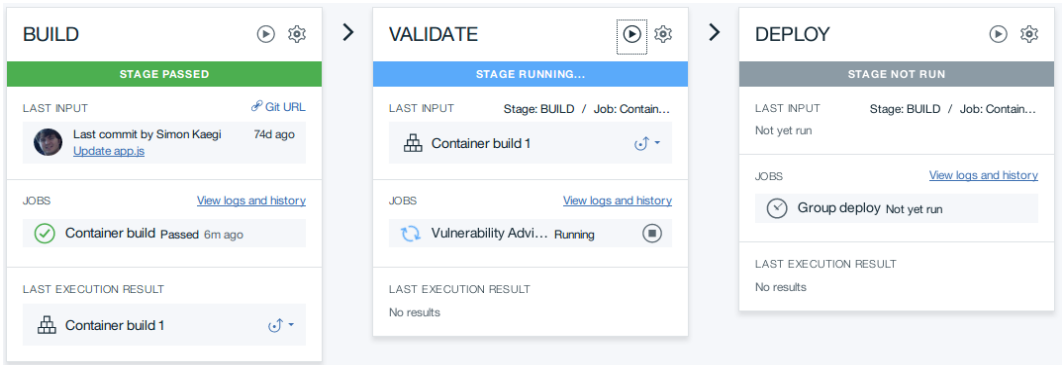

Illustration 56: Validation Stage Running

4.8.6 Validation Stage Fails and Continues to Deploy

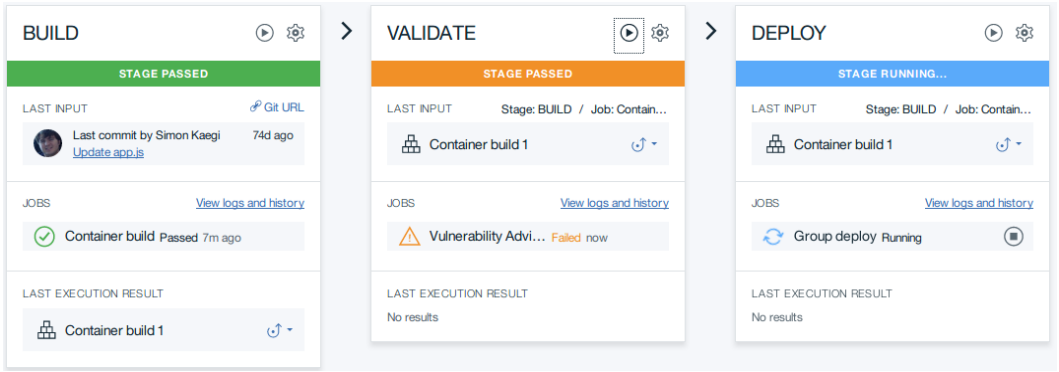

Illustration 57: Continue to Deploy

4.8.7 Successful Deployment

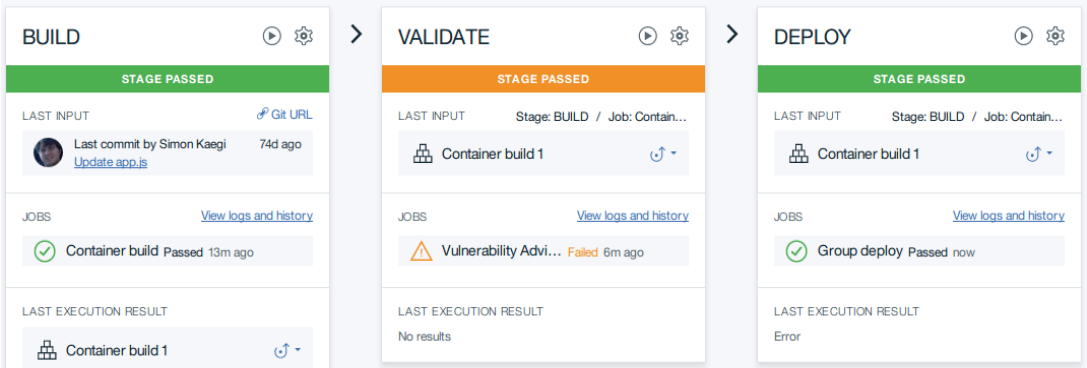

Illustration 58: Successful Deployment

5 Project Management

5.1 Strategy

The project plan focuses more on the problems that need to be solved immediately and omits details such as time series storage and Amalgam8 microservices integration with OpenWhisk. These problems will be investigated in the 3rd iteration provided all higher priority tasks are completed on time.

The time to complete each task in a single iteration is roughly estimated and requires closer examination. Time will also be invested into migrating from Trello project management to the IBM DevOps services. This service provides a way of adding tickets to an iteration and adding developers to a ticket. This skill will then be combined with Git version control skills which will provide the ability to fix bugs on a collaborated project. After completing a task, other developers that may be depending on that task finish will be notified. This process dramatically speeds up collaborated development time and reduces duplicated effort.

Trello is straight forward and familiar so it will remain a core part of the project management strategy. The process is based on the kanban strategy, where tasks flow from left to right as they go from start to finish. The ability to easily move tasks from one list to another as task priority changes will allow for more project flexibility.

5.2 Trello Project Labels

Labels are used in order to identify the context of each task.

Illustration 59: Project Labels

5.3 Project Iteration Planning

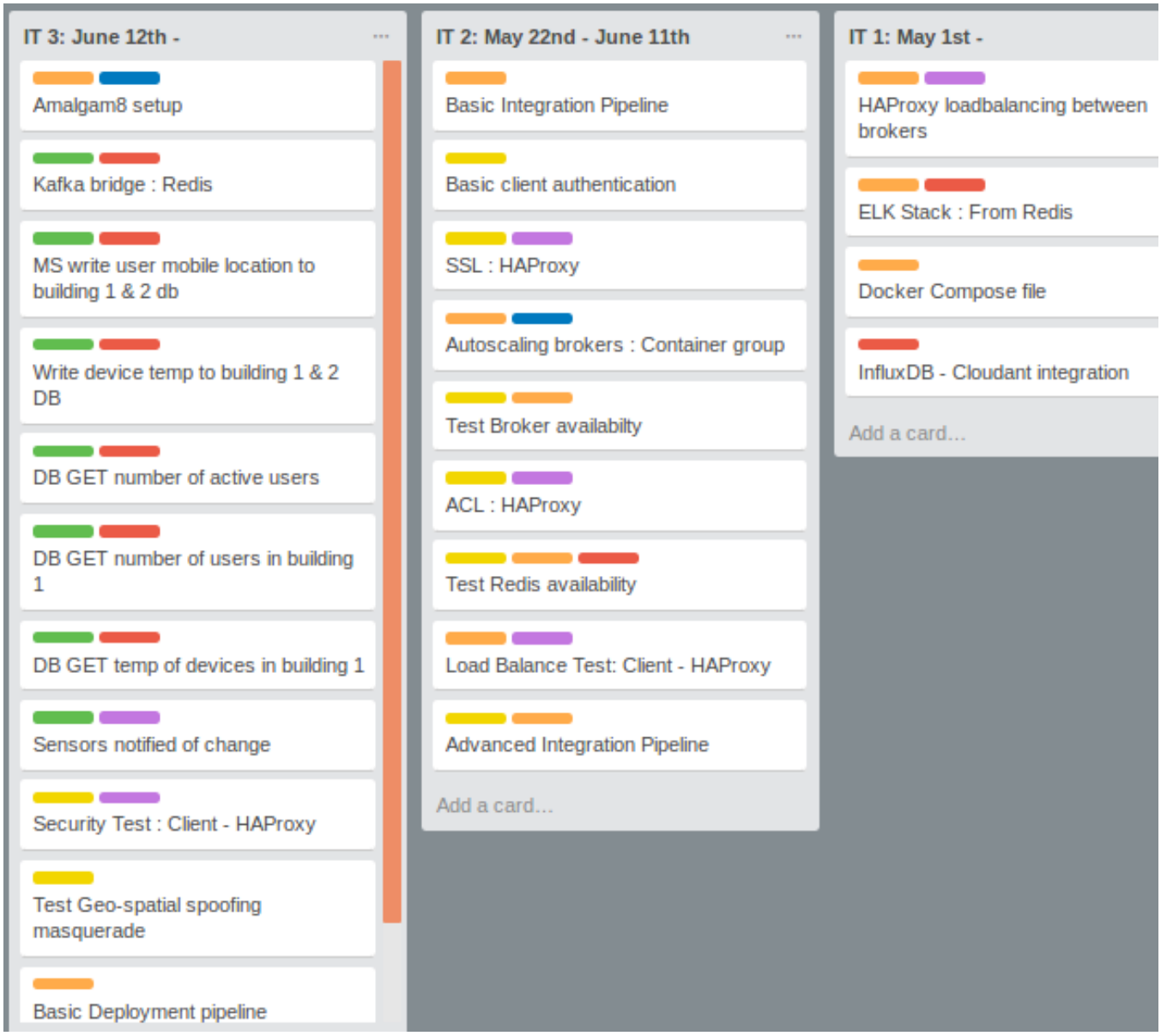

Illustration 60: Trello Iteration Planning

5.4 Iteration 1: May 1st – May 21st

The deliverable is focused towards load balancing between brokers. The lower priorities being the Redis and Logstash caching of data. It will be necessary to write a Docker Compose file that will orchestrate the various Containers together. InfluxDB will be integrated with Cloudant for creating time-series data.

5.5 Iteration 2: May 22nd – June 11th

With the core functionality implemented, work will be directed towards setting up a pipeline that will continuously test the code any time a change is made. Basic tests will then be written to test the authentication of devices.

With load balancing working successfully between two Mosca brokers, attention will be directed towards the problem of the Container Scalable Group. Specifically the problem of the group not being able to expose port 1883.

Security features such as SSL and ACL will be implemented for the HAProxy Container.

Load balance testing will be carried out on the HAProxy, Mosca and Redis Containers. At this point the continuous integration pipeline will need to be reviewed.

5.6 Iteration 3: June 12th – July 2nd

With a solid foundation setup for IoT and testing, focus will be directed towards Amalgam8 which provides capabilities such as systematic resiliency testing, red/black deployment, and canary testing necessary for rapid experimentations and insight.

Microservices for storing and retrieving data based on location will be implemented using Amalgam8. The goal here is to provide future students with both an API and a DevOps tool that they can work with in order to create smart campus applications for the university.

A brief investigation into Geospatial location spoofing.

Finally an investigation into how continuous deployment can be achieved.

5.7 Iteration 4: July 3rd – July 23rd

Deliverable will be a demonstration of canary testing and red/black deployment using Amalgam8. It is expected that many objectives from Iteration 3 will flow over into Iteration 4. The remaining time for the project must be dedicated to writing up the final theses report.

6 Progress to Date

6.1 Christmas Break Summary

The work completed during the Christmas break proved to be somewhat wasteful. More time should have been spent on topics such as load balancing, MQQT client simulation/authentication, brokers, pub/sub protocols.

A basic front-end application based on Angular.js was specified in the Autumn report. The application was added to the scope of the project in order to make the final product more demonstrable. The original application was specified to have a basic web page that would display a map of the UL campus with the location of the active users of the mobile application. This lead to the discovery of the Ionic framework which provides a foundation for developing a mobile application using Angular.js. After returning from the Christmas break, the project supervisor advised against taking this new direction as it dramatically increased the scope of the project far beyond the time allotted.

The second area of work revolved around the area of DevOps. This consisted of learning the basics of the Docker CLI as well as networking/orchestrating Containers using the Docker Compose tool.

The Jenkins build tool was studied for continuous integration, it is a DevOps tool for automating repetitive tasks. A tutorial was completed on how to implement a “web hook” to a GitHub code repository. Any changes to the code in the repository triggered a “Jenkins job” which simply ran a Bash script that echoed “hello world”. It turns out Bluemix provides everything a basic DevOps Engineer would need for a CI/CD pipeline, in fact the the IBM DevOps continuous delivery pipeline is using Jenkins behind the scenes. This removes the pains of installing and provisioning a Jenkins server.

The third area of work revolved around learning the basics of Node.js as no prior knowledge of Javascript was known before undertaking the project. This study was done in a copy book but has not yet been documented on GitHub.

In summary, the work done over the Christmas break was somewhat wasteful but nevertheless provided the necessary DevOps skills and Node.js background knowledge to go about developing and designing the proposed architecture as specified in this Spring report.

6.2 Week 1 Log

6.2.1 Week 1 Time Bar Chart

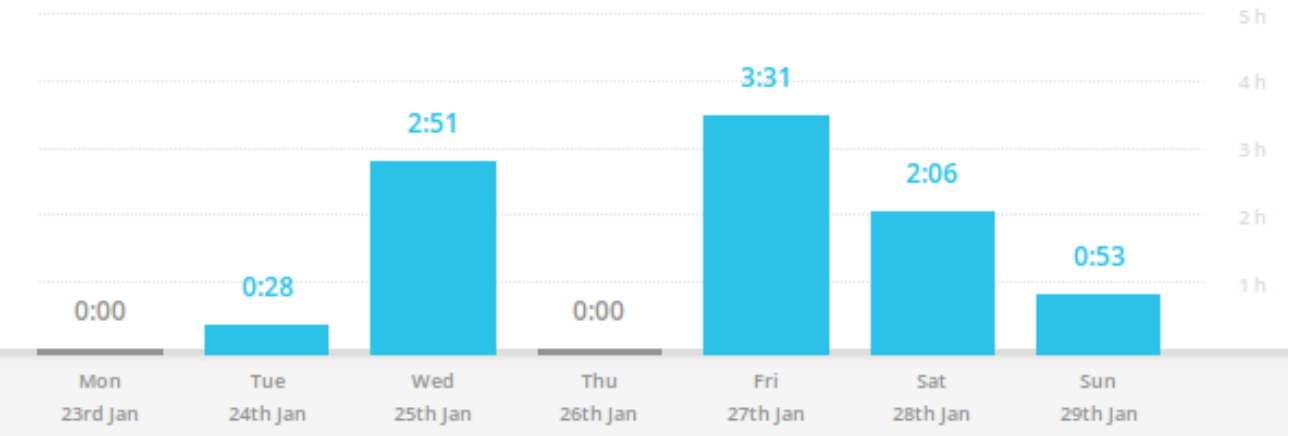

Illustration 61: Week 1 Bar Chart

6.2.2 Tasks Completed

- Meet with supervisor [21]
- **Report Template** [21]
- **Research MQTT** [21]
- **Research Kafka** [21]
- **Find similar project** [21]
- **Research databases** [21]

6.2.3 Week 1 Log Entries

Entry 24/01/17:

"Today I met with my supervisor. We discussed various aspects of the project. My supervisor advised me to focus on the back-end architecture for now as the scope of the project is too large."[21]

Entry 25/01/17:

"Today I ran into some problems setting up the academic free trial. I had to email the Bluemix support team in order to resolve the issue."[21]

6.2.4 Week 1 Time Log

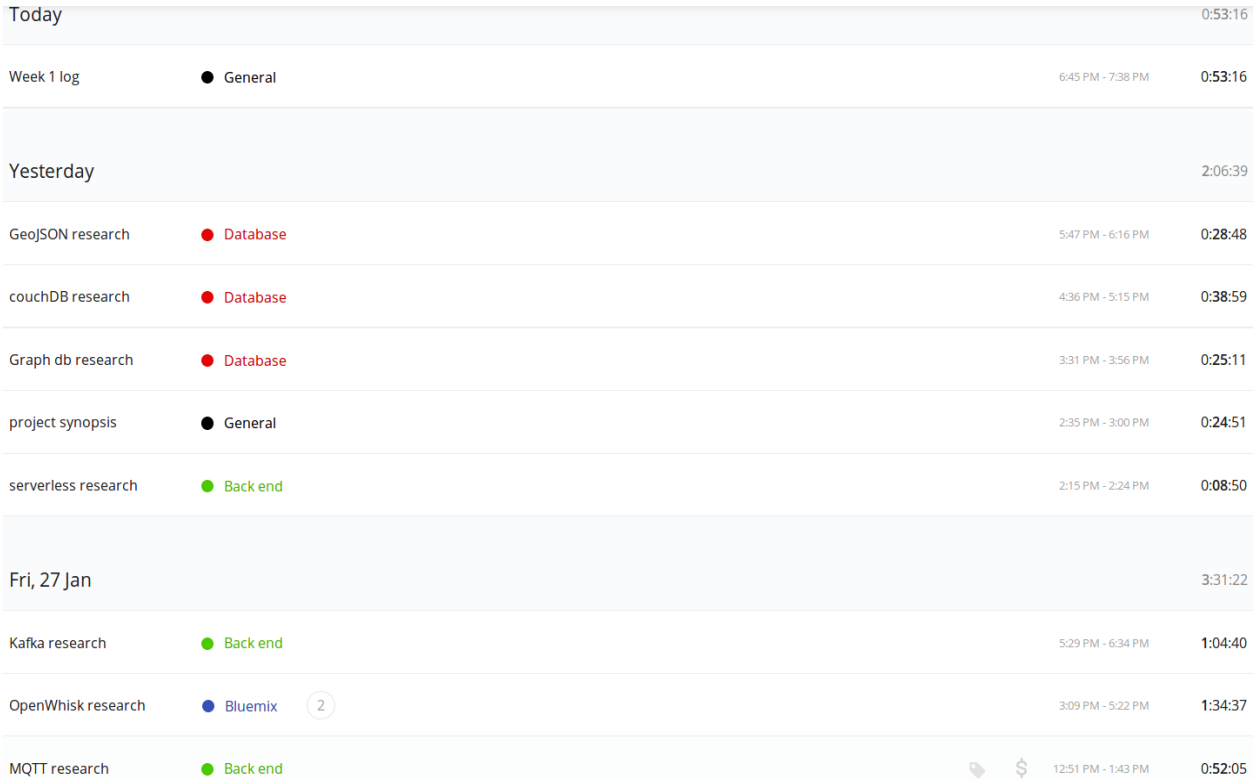

Illustration 62: Week 1 Time Log 2

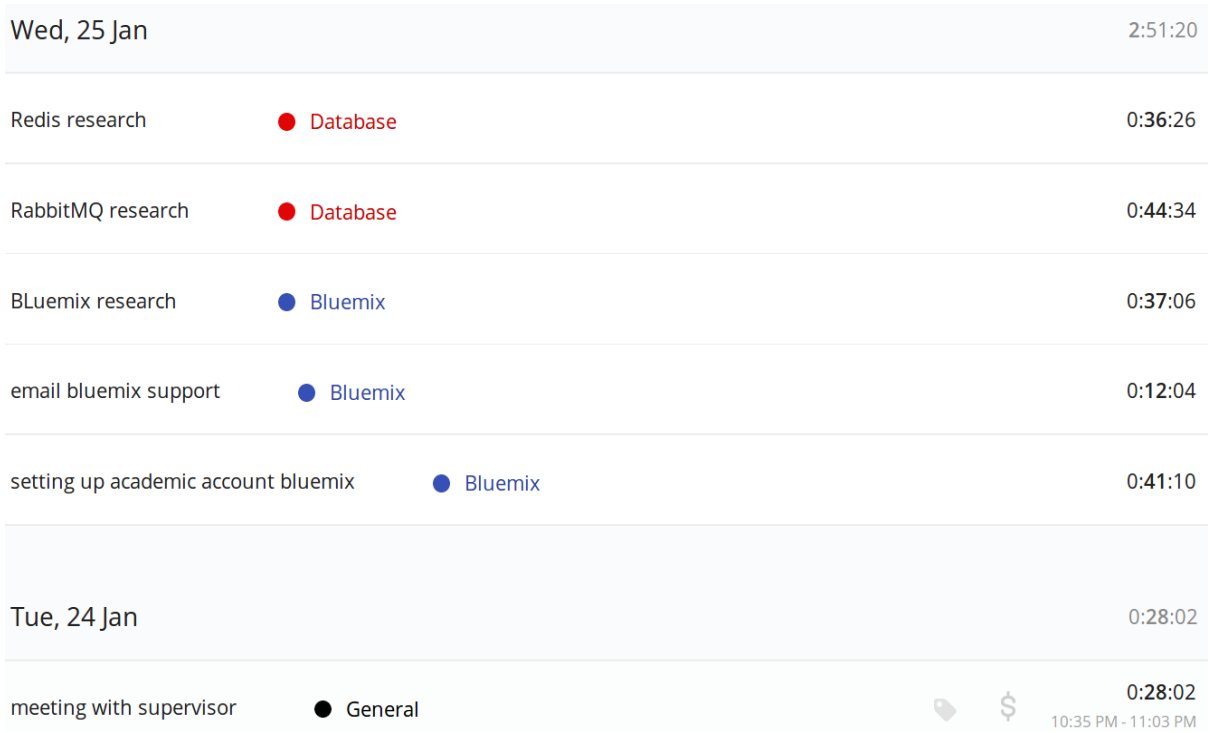

Illustration 63: Week 1 Time Log 1

6.3 Week 2 Log

6.3.1 Week 2 Time Bar Chart

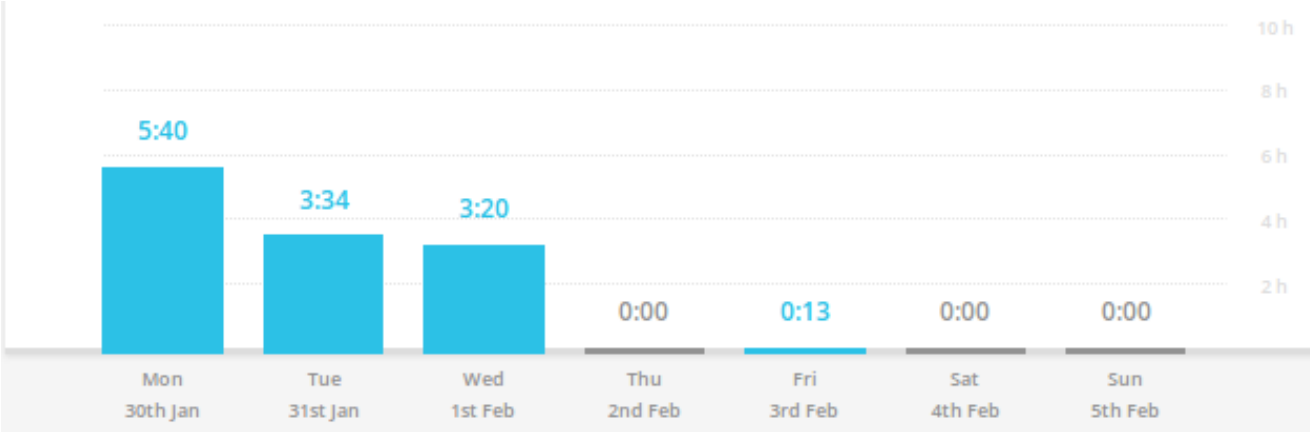

Illustration 64: Week 2 Bar Chart

6.3.2 Tasks Completed

- **Create new Bluemix account for academic free trial** [22]
- **IBM Container Service research** [22]
- **IBM Container group research** [22]
- **Amalgam8 research** [22]
- **General research (jumping between mqtt brokers and bluemix links)** [22]
- **MQTT container research** [22]
- **Microservices from Theory to Practice (IBM document)** [22]
- **Mosca research** [22]
- **Questions & answers (self evaluation)** [22]
- **Research links document for supervisor** [22]
- **(rough) architectural diagram** [22]

6.3.3 Week 2 Time Log

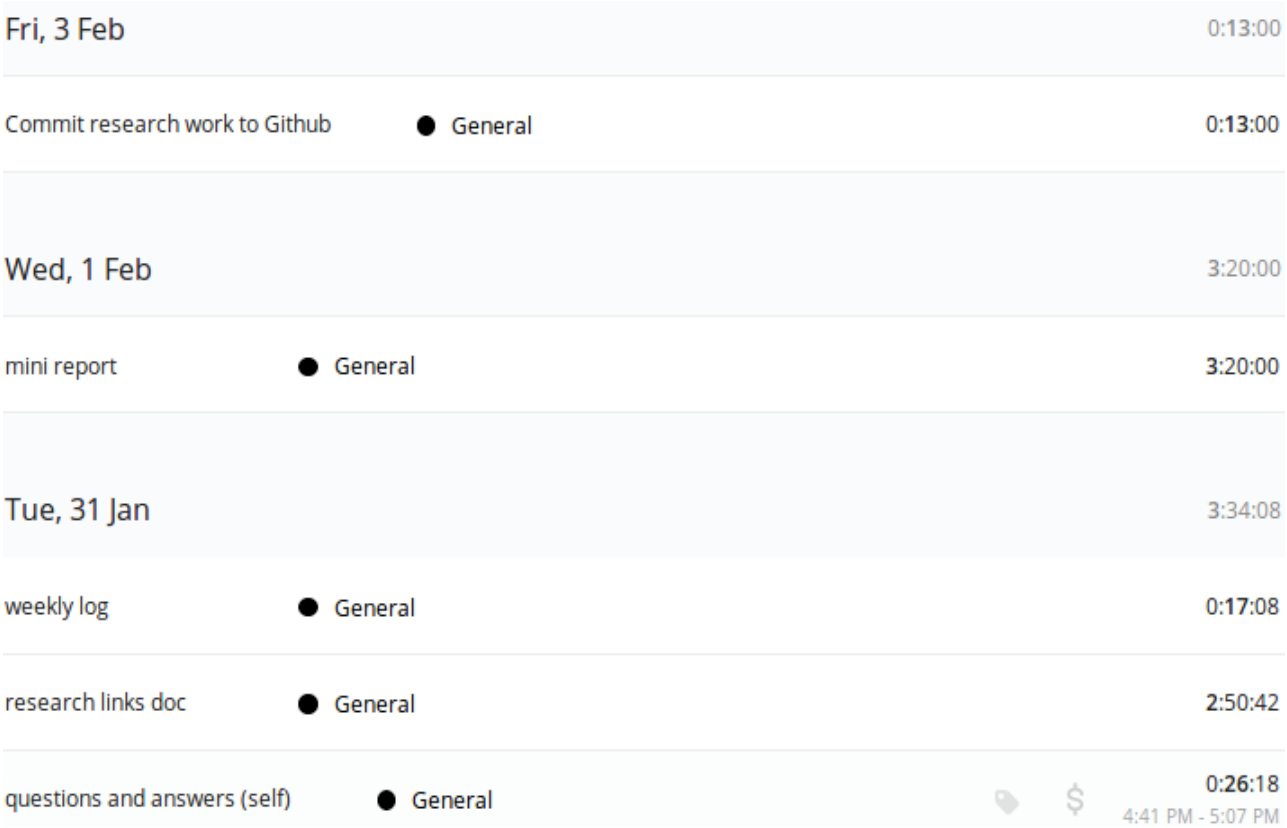

Illustration 65: Week 2 Time Log 2

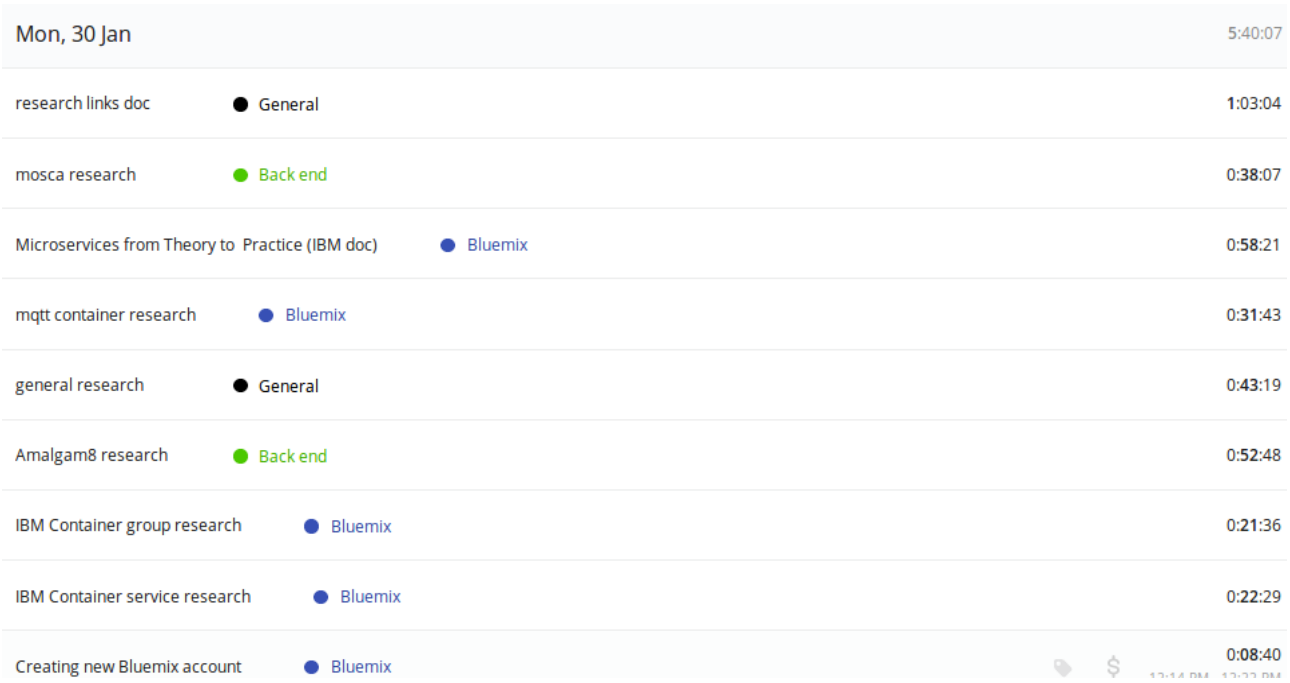

Illustration 66: Week 2 Time Log 1

6.4 Week 3 Log

6.4.1 Week 3 Time Bar Chart

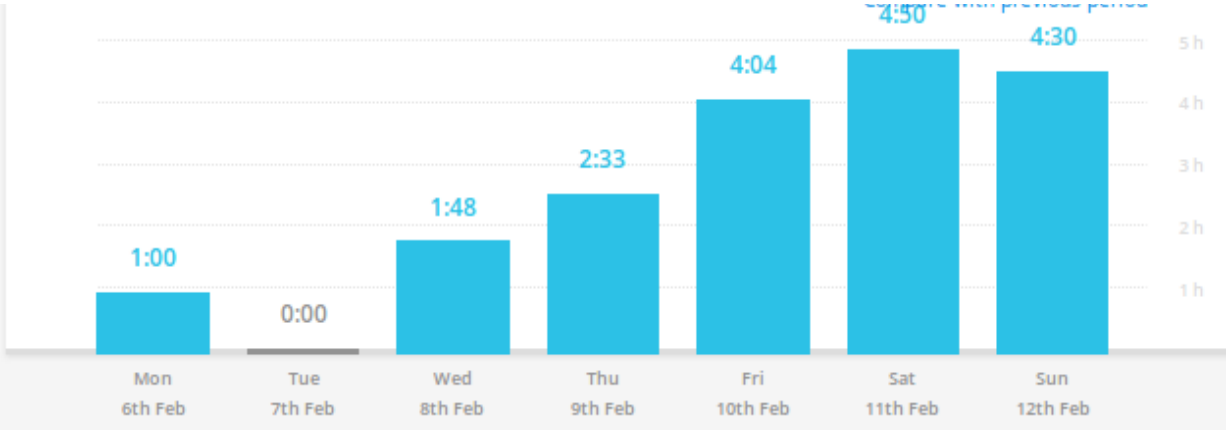

Illustration 67: Week 3 Bar Chart

6.4.2 Week 3 Time Log

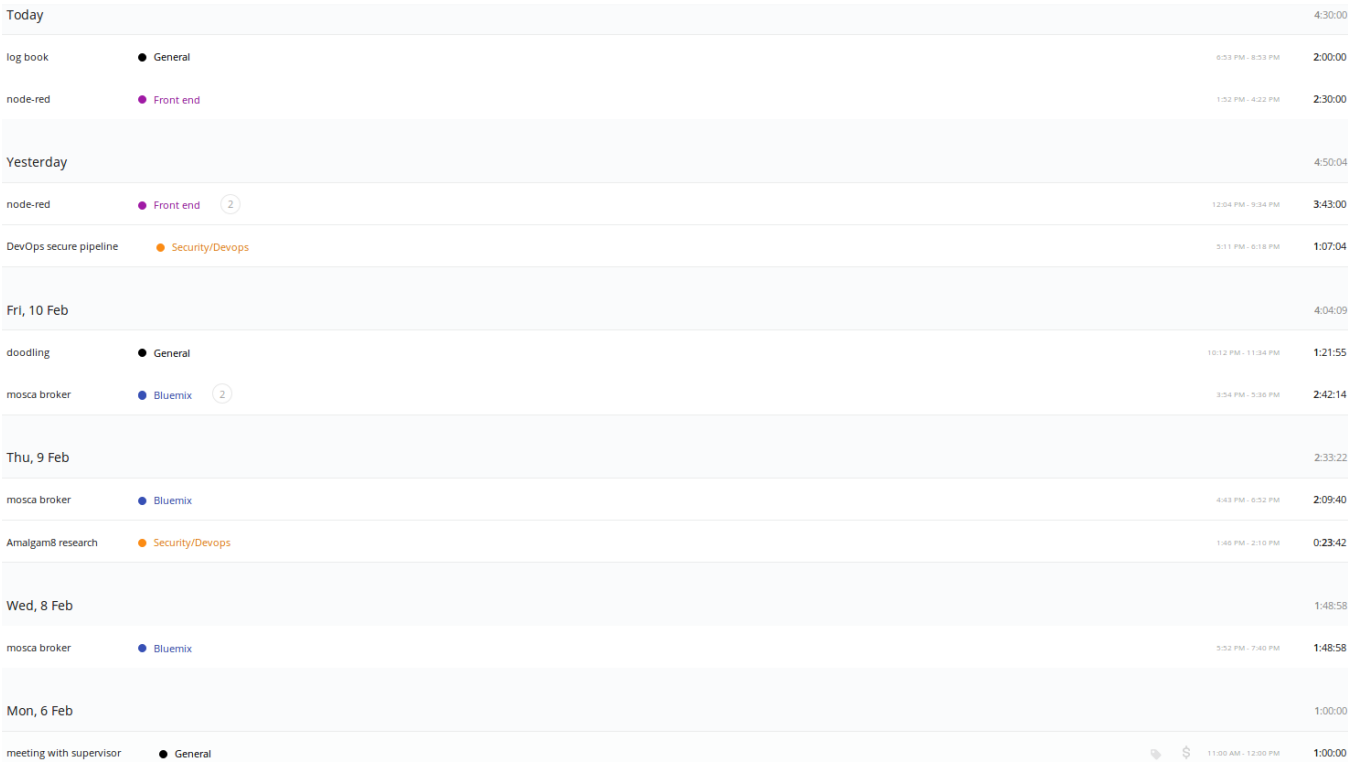

Illustration 68: Week 3 Time Log

6.4.3 Tasks completed

- **Mosca Broker working locally** [23]
- **Mosca Broker on Bluemix** [23]
- **Node-RED injects msgs** [23]
- **Node-RED sends to broker on bluemix** [23]
- **Node-RED encodes and decodes base 64** [23]
- **Node-RED stores data in cloudant** [23]
- **Security DevOps pipeline tutorial** [23]

6.4.3 Week 3 Log Entries

Entry 08/01/17:

"While still recovering from a flu, I managed to familiarize myself with the cloud foundry Bluemix CLI. I made my goal to simply get a custom container image onto Bluemix. I ran into various problems such as logging into the american "ng" domain and not the "eu-gb" europe domain. Half of the delay was due to poor memory of how I did this last time. The other half being my state of mind (illness)."[23]

Entry 09/01/17:

"Today I managed to get a mosca broker running on my local Docker host. Using Mosquitto service to pub and sub."[23]

Entry 10/01/17:

"Today I deployed a mosca broker on IBM containers. I carried out the same test I did locally."[23]

Entry 11/01/17:

"Today I used Node-RED for the first time properly. I managed to get data flowing from an inject node to my broker running on IBM Container on Bluemix."[23]

Entry 12/01/17:

"Today I managed to encode/decode base 64 using Node-RED. I also managed to get the decoded data into a cloudant database. I ran into many problems this week, but time invested carried me through."[23]

6.5 Week 4 Log

6.5.1 Week 4 Time Bar Chart

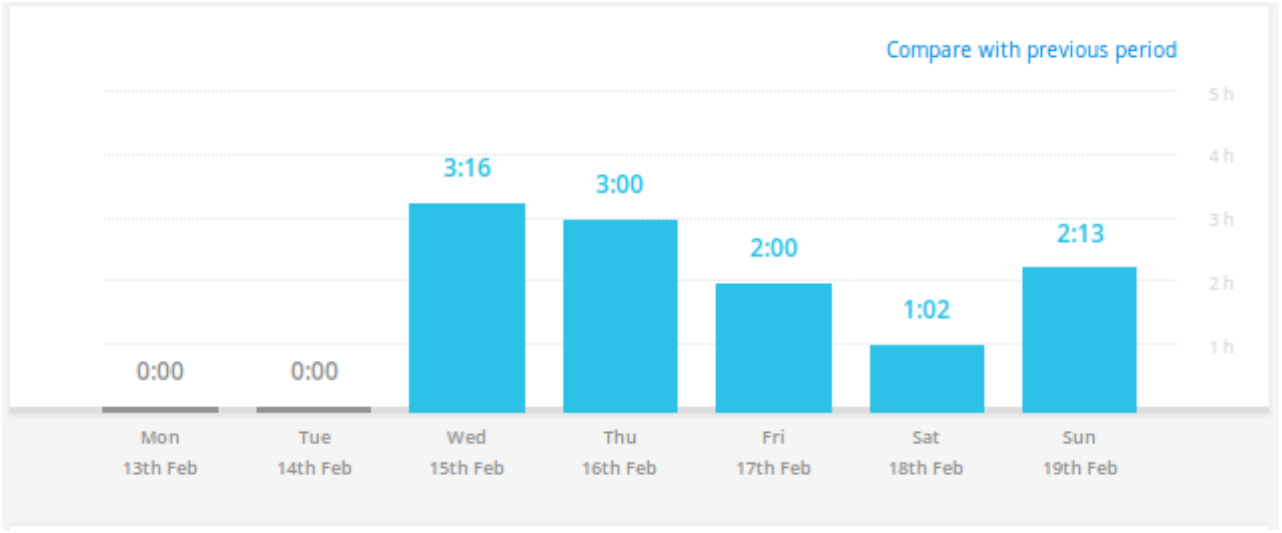

Illustration 69: Week 4 Bar Chart

6.5.2 Week 4 Time Log

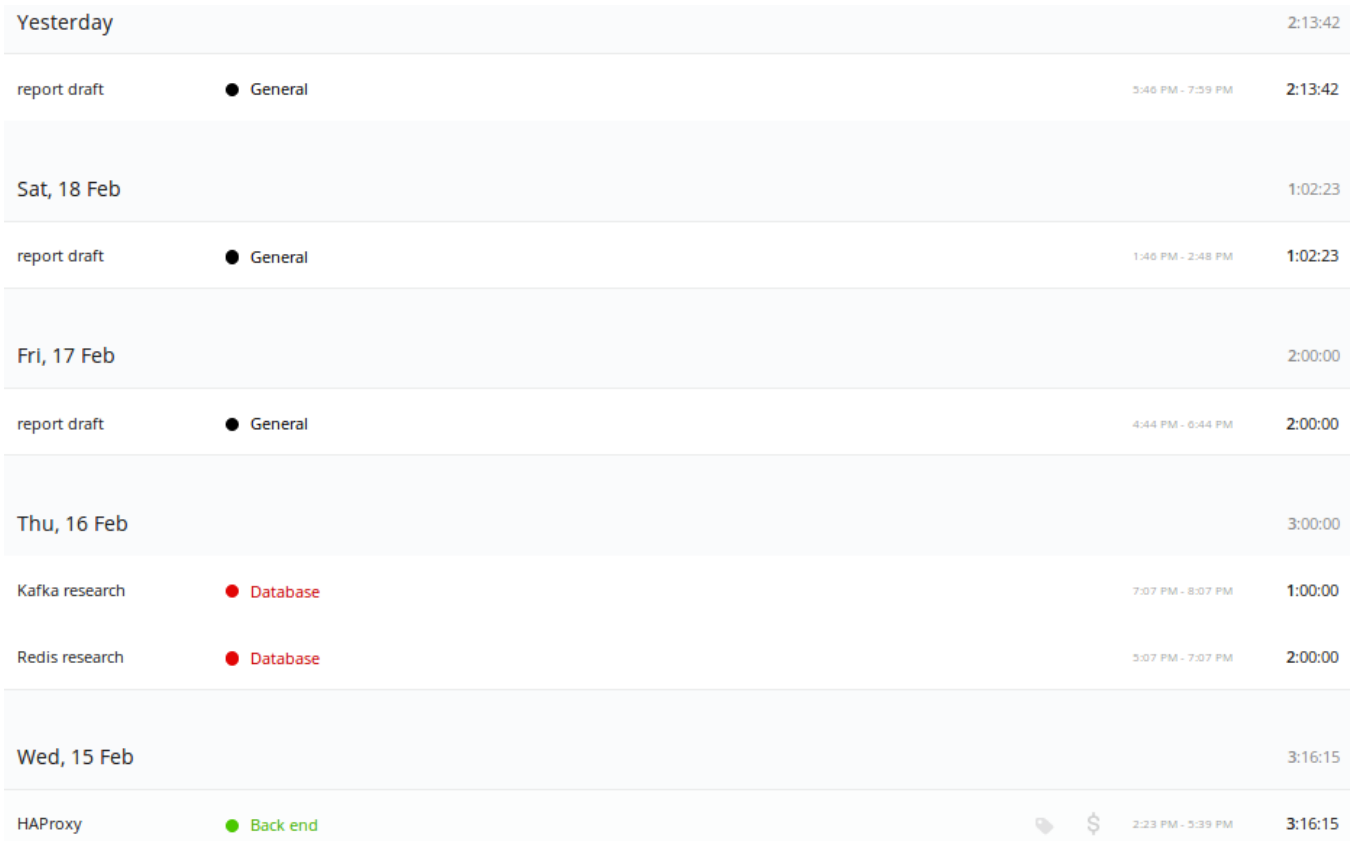

Illustration 70: Week 4 Time Log

6.5.3 Tasks Completed

- **Attempted to implement HAProxy container** [24]
- **Redis research** [24]
- **Kafka research** [24]
- **ELK Stack research** [24]
- **report draft** [24]
- **Literature survey spring report** [24]

6.5.4 Week 4 Log Entries

Entry 15/02/17:

"Today I struggled to get the HAProxy container working. I spent the first hour and a half just trying to attach an "overide config" as a volume to the container. Once "**docker logs <haproxy container id>**" stopped reporting errors, I knew at least the container was building successfully. The next step is to export HAProxy logs in order to find out more information as to what the problem might be."[24]

Entry 16/02/17:

"Today I researched both Kafka and Redis, trying to understand the difference between them and why/where you would use one over the other."[24]

Entry 17/02/17:

"Today I started my Spring report, this consisted of laying out the template and putting in section headings. I also research Amalgam8 microservices framework."[24]

Entry 18/02/17:

"Today I worked on "section 2 Literature survey" of my spring report. I also researched container monitoring and logging using the ELK stack."[24]

Entry 19/02/17:

"Today I continued work on my Spring report, finishing off the literature survey section."[24]

7 Evaluation

7.1 Database Comparison

Editorial information provided by DB-Engines				
Name	**Cloudant** X	**Elasticsearch** X	**InfluxDB** X	**Redis** X
Description	Database as a Service offering based on Apache CouchDB	A modern search and analytics engine based on Apache Lucene	DBMS for storing time series, events and metrics	In-memory data structure store, used as database, cache and message broker
Database model	Document store	Search engine	Time Series DBMS	Key-value store
DB-Engines Ranking Trend Chart	Score 5.36 Rank #47 Overall #8 Document stores	Score 108.31 Rank #11 Overall #1 Search engines	Score 6.77 Rank #43 Overall #1 Time Series DBMS	Score 114.03 Rank #10 Overall #1 Key-value stores
Website	cloudant.com	www.elastic.co/products/-elasticsearch	www.influxdata.com/time-series-platform/influxdb	redis.io
Technical documentation	docs.cloudant.com	www.elastic.co/guide	docs.influxdata.com/influxdb	redis.io/documentation
Developer	IBM, Apache Software Foundation	Elastic		Salvatore Sanfilippo
Initial release	2010	2010	2013	2009
Current release		5.2.1, February 2017	v1.1.1, December 2016	3.2.8, February 2017
License	commercial	Open Source	Open Source	Open Source
Cloud-based	yes	no	no	no
Implementation language	Erlang	Java	Go	C
Server operating systems	hosted	All OS with a Java VM	Linux OS X	BSD Linux OS X Windows
Data scheme	schema-free	schema-free	schema-free	schema-free
Typing	no	yes	Numeric data and Strings	partial
XML support	no		no	no
Secondary indexes	yes	yes	no	no
SQL	no	no	no	no
APIs and other access methods	RESTful HTTP/JSON API	Java API RESTful HTTP/JSON API	HTTP API JSON over UDP	proprietary protocol

Illustration 71: db-engines 1

Server-side scripts	View functions (Map-Reduce) in JavaScript	yes	no	Lua
Triggers	yes	yes	no	no
Partitioning methods	Sharding	Sharding	Sharding	Sharding
Replication methods	Master-master replication Master-slave replication	yes	selectable replication factor	Master-slave replication
MapReduce	yes	no	no	no
Consistency concepts	Eventual Consistency	Eventual Consistency		Eventual Consistency
Foreign keys	no	no	no	no
Transaction concepts	no	no	no	Optimistic locking, atomic execution of commands blocks and scripts
Concurrency	yes	yes	yes	yes
Durability	yes	yes	yes	yes
In-memory capabilities	no		yes	yes
User concepts	Access rights for users can be defined per database		simple rights management via user accounts	Simple password-based access control

Illustration 72: db-engines 2

[30]

7.2 Container Memory Limitations

The memory limit for deploying Containers in a single name space is 2GB using the academic free-trial on Bluemix.

Budget:

1x HAProxy = 256MB

2x Mosca Brokers = 256MB * 2 = 512MB

1x Redis = 256MB

1x InfluxDB = 256MB

Total Used:

1.28 GB

7.3 Considerations

Investigation into to the minimum memory required for each Container will need to be worked out. It seam feasible to implement the overall architecture of the project without a need for additional funds. However in order to perform load balance tests, both the memory and financial budget will need to be reviewed.

Currently the limit for public IP addresses is 2 in one namespace. The HAProxy Container will be configured accordingly to handle this issue, acting as the gateway to the back-end services.

InfluxDB for time-series data was a late addition to the project scope. It may need to be postponed if higher priority tasks are not completed in time. The plan is to trigger an Openwhisk action whenever a write to the Cloudant database is made. The action will organize the data into a geospatial group (e.g Schuman Building) before commiting to the InfluxDB for time-series data.

It is paramount to continue work on the project as soon as exams are over in order to realistically complete all objectives on time.

Lastly the bulk of the research carried out during this semester was compiled into two pdfs titled *research-links.pdf* and *Theses-Summary.pdf*. These documents can be found in the “Extra work” directory on the GitHub repository.[34] These documents contain a more informal approach to researching the various aspects of the project.

8 Bibliography

[1]T. Flynn, "GitHub - 16117743/INS-Thesis-Documentation: Documentation of Information & Network Security MEng Thesis", *Github.com*, 2017. [Online]. Available: https://github.com/16117743/INS-Thesis-Documentation. [Accessed: 23- Feb- 2017].

[2]M. Collina, "An Internet of Things System - How To Build It Faster", *nearForm*, 2017. [Online]. Available: http://www.nearform.com/nodecrunch/internet-of-things-how-to-build-it-faster/. [Accessed: 23- Feb- 2017].

[3]"HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer", *Haproxy.org*, 2017. [Online]. Available: http://www.haproxy.org/. [Accessed: 23- Feb- 2017].

[4]J. Wu, "A Network Function Virtualization based Load Balancer for TCP", Master of Science, ARIZONA STATE UNIVERSITY, 2017.

[5]M. Collina, "GitHub - mcollina/mosca: MQTT broker as a module", *Github.com*, 2017. [Online]. Available: https://github.com/mcollina/mosca. [Accessed: 23- Feb- 2017].

[6]"RabbitMQ - Messaging that just works", *Rabbitmq.com*, 2017. [Online]. Available: https://www.rabbitmq.com/. [Accessed: 23- Feb- 2017].

[7]"mqtt", *npm*, 2017. [Online]. Available: https://www.npmjs.com/package/mqtt. [Accessed: 23- Feb- 2017].

[8]"Redis", *Redis.io*, 2017. [Online]. Available: https://redis.io/. [Accessed: 23- Feb- 2017].

[9]"Kafka vs. Redis: Log Aggregation Capabilities and Performance", *Logz.io*, 2017. [Online]. Available: http://logz.io/blog/kafka-vs-redis/. [Accessed: 23- Feb- 2017].

[10]"Kafka", 2017. [Online]. Available: https://kafka.apache.org/. [Accessed: 23- Feb- 2017].

[11]"Elasticsearch: RESTful, Distributed Search & Analytics | Elastic", *Elastic.co*, 2017. [Online]. Available: https://www.elastic.co/products/elasticsearch. [Accessed: 23- Feb- 2017].

[12]"Logstash: Collect, Parse, Transform Logs | Elastic", *Elastic.co*, 2017. [Online]. Available: https://www.elastic.co/products/logstash. [Accessed: 23- Feb- 2017].

[13]"Kibana: Explore, Visualize, Discover Data | Elastic", *Elastic.co*, 2017. [Online]. Available: https://www.elastic.co/products/kibana. [Accessed: 23- Feb- 2017].

[14]"Bluemix Container monitoring and logging", *Console.ng.bluemix.net*, 2017. [Online]. Available: https://console.ng.bluemix.net/docs/containers/monitoringandlogging/container_ml_ov.html. [Accessed: 23- Feb- 2017].

[15]A. AS, "Visualizing Docker Compose with Ardoq - Ardoq Blog", *Ardoq*, 2017. [Online]. Available: https://ardoq.com/visualizing-docker-compose/. [Accessed: 23- Feb- 2017].

[16]"Server-side service discovery pattern", *Microservices.io*, 2017. [Online]. Available: http://microservices.io/patterns/server-side-discovery.html. [Accessed: 23- Feb- 2017].

[17]E. Norman and E. Norman, "Amalgam8: Framework for composition and orchestration of microservices - Bluemix Blog", *Bluemix Blog*, 2017. [Online]. Available: https://www.ibm.com/blogs/bluemix/2016/07/amalgam8-framework-for-microservices-orchestration/?S_TACT=M16103KW. [Accessed: 23- Feb- 2017].

[18]"practice continuous integration", *Ibm.com*, 2017. [Online]. Available: https://www.ibm.com/devops/method/content/code/practice_continuous_integration/. [Accessed: 23- Feb- 2017].

[19]"TRANSIT: Flexible pipeline for IoT data with Bluemix and OpenWhisk – OpenWhisk", *Medium*, 2017. [Online]. Available: https://medium.com/openwhisk/transit-flexible-pipeline-for-iot-data-with-bluemix-and-openwhisk-4824cf20f1e0#.kv9gk4j2g. [Accessed: 23- Feb- 2017].

[20]"How to Build an High Availability MQTT Cluster for the Internet of Things", *Medium*, 2017. [Online]. Available: https://medium.com/@lelylan/how-to-build-an-high-availability-mqtt-cluster-for-the-internet-of-things-8011a06bd000#.iku8iazv6. [Accessed: 23- Feb- 2017].

[21]T. Flynn, "Spring Week 1 Log", 2017. [Online]. Available: https://github.com/16117743/INS-Thesis-Documentation/blob/master/Spring%20logs/Spring_Wk1.pdf. [Accessed: 23- Feb- 2017].

[22]T. Flynn, "Spring Week 2 Log", 2017. [Online]. Available: https://github.com/16117743/INS-Thesis-Documentation/blob/master/Spring%20logs/Spring_Wk2.pdf. [Accessed: 23- Feb- 2017].

[23]T. Flynn, "Spring Week 3 Log", 2017. [Online]. Available: https://github.com/16117743/INS-Thesis-Documentation/blob/master/Spring%20logs/Spring_Wk3.pdf. [Accessed: 23- Feb- 2017].

[24]T. Flynn, "Spring Week 4 Log", 2017. [Online]. Available: https://github.com/16117743/INS-Thesis-Documentation/blob/master/Spring%20logs/Spring_Wk4.pdf. [Accessed: 23- Feb- 2017].

[25]"InfluxData Documentation", *Docs.influxdata.com*, 2017. [Online]. Available: https://docs.influxdata.com/influxdb/v0.8/api/continuous_queries/. [Accessed: 25- Feb- 2017].

[26]"GitHub - influxdata/influxdb: Scalable datastore for metrics, events, and real-time analytics", *Github.com*, 2017. [Online]. Available: https://github.com/influxdata/influxdb. [Accessed: 25- Feb- 2017].

[27]"DevOps Automation Cookbook", *Google Books*, 2017. [Online]. Available: https://books.google.ie/books?id=k_SoCwAAQBAJ&pg=PA176&redir_esc=y#v=onepage&q&f=false. [Accessed: 25- Feb- 2017].

[28]"InfluxDB", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/InfluxDB. [Accessed: 25- Feb- 2017].

[29]"InfluxData Documentation", *Docs.influxdata.com*, 2017. [Online]. Available: https://docs.influxdata.com/influxdb/v0.8/api/continuous_queries/. [Accessed: 25- Feb- 2017].

[30]"Cloudant vs. Elasticsearch vs. InfluxDB vs. Redis Comparison", *Db-engines.com*, 2017. [Online]. Available: http://db-engines.com/en/system/Cloudant%3BElasticsearch%3BInfluxDB%3BRedis. [Accessed: 27- Feb- 2017].

[31]"Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach", *Google Books*, 2017. [Online]. Available: https://books.google.ie/books?id=eOZyCgAAQBAJ&pg=PA95&lpg=PA95&dq=ibm+mqtt+mirroring&source=bl&ots=sggWe9DQfH&sig=5j7UvNldc8Hgon7L8mDC45_09lc&hl=en&sa=X&ved=0ahUKEwj5q6G1r7HSAhVoCcAKHXcwAtIQ6AEIIjAB#v=onepage&q=ibm%20mqtt%20mirroring&f=false. [Accessed: 27- Feb- 2017].

[32]"REST and MQTT: Yin and Yang of Micro-Service APIs", *Dejan Glozic*, 2017. [Online]. Available: https://dejanglozic.com/2014/05/06/rest-and-mqtt-yin-and-yang-of-micro-service-apis/. [Accessed: 27- Feb- 2017].

[33]M. Koster, M. Koster and V. profile, "Event Models for RESTful APIs", *Iot-datamodels.blogspot.ie*, 2017. [Online]. Available: http://iot-datamodels.blogspot.ie/2013/05/event-models-for-restful-apis.html. [Accessed: 28- Feb- 2017].

[34]T. Flynn, "Spring Week 4 Log", 2017. [Online]. Available: https://github.com/16117743/INS-Thesis-Documentation/tree/master/Extra%20work. [Accessed: 23- Feb- 2017].

[35]"Authentication & Authorization · mcollina/mosca Wiki · GitHub", *Github.com*, 2017. [Online]. Available: https://github.com/mcollina/mosca/wiki/Authentication-&-Authorization. [Accessed: 01- Mar- 2017].