

High-throughput Serverless Computing

Supervisor: Dr. Thomas Heinis

Taixi Pan
tp1119

Javier Garcia
jg2121

Stefan Neamtu
sn1319

Yuang Li
yl4519

Feifan Fan
ff519

January 10, 2022

Contents

1	Executive Summary	3
2	Introduction	3
2.1	Motivation	3
2.2	Objectives	3
2.2.1	Target 1 (Interactive Tracking Interface)	4
2.2.2	Target 2 (Simulation Engine)	4
2.3	Achievements	4
3	Design and Implementation	4
3.1	Overall Design	4
3.2	Interactive Tracking Interface	4
3.2.1	User Upload Handler: AWS IoT Core	4
3.2.2	Data persistence: AWS Location Service Trackers	5
3.2.3	Data Fetching: AWS Lambda	7
3.2.4	User Interface: React App	8
3.3	Simulation Engine	8
3.3.1	Generating the necessary data	9
3.3.2	Parsing the data, updating the offset	9
3.3.3	Sending the location updates	10
3.4	Analyse Simulation Results	11
3.4.1	The Simulation Log Analyser	11
4	Evaluation	13
4.1	Unit Testing For Each Component	13
4.2	Integration Test	14
4.3	CI/CD Pipeline	14
4.4	Simulation engine	14
4.5	User Feedback	16
5	Ethical Considerations	16
	References	17

List of Figures

1	Overall Architecture of Design	5
2	Trigger for an AWS Lambda [1] function	5
3	Device positions stored in a tracker	6
4	An Example of API Gateway [2] Pipeline	7
5	A DOM generated by a simple HTML application [3]	8
6	Concept map describing our Akka Actor model	11
7	Sample simulation log	12
8	Finding the timestamp difference in matched entries	12
9	Mock test for IoT Device Creation Script	13
10	Pressure test of the analyser	13
11	AWS MQTT test client	14
12	Lambda metrics for a 30-device simulation with the actor model	15
13	Lambda metrics for a 30-device simulation with the task array model (15 sim. files and 2 devices per file)	15
14	An IoT thing policy	16

1 Executive Summary

Serverless architecture, unlike server-ful architecture, frees the developer from the need of managing a server manually. Instead, the developers can focus on how each part of the application should behave, and how each function or method would interact, leaving the jobs (e.g. maintaining a server or allocating resources for running modularised tasks) to cloud providers.

With a serverless architecture, scaling up of an application is automatic for the developers and we could expect the behaviour of applications with different scales in benchmarking, especially with high throughput. However, the higher throughput doesn't mean a greater cost for allocating more resources with serverless architecture. The paid resources are allocated only when the functions are invoked. Serverless would be the next generation architectural pattern, just like how containers superseded physical servers on the cloud.

To investigate and evaluate the potential of serverless architecture, we developed a simple delivery tracking system, in which uploaded real-time positions can be viewed by the users. We then simulated many concurrent uploads to the system and try to see how it performs.

Our project makes use of [Amazon Web Services](#) [4] to implement the core functionality of a delivery tracking app: uploading and retrieving device locations. With the serverless design, one can easily extend it to allow for tens of thousands of users.

Our project also features a simulation engine that can simulate thousands of devices uploading location data concurrently. The uploaded data can be viewed from our user web application and the performance statistics for one round of simulation can also be recorded. This project would be beneficial for those who wish to evaluate the potential of adopting a serverless architecture in building a delivery tracking app that would expect a large number of users.

2 Introduction

2.1 Motivation

Delivery tracking is a typical situation where highly concurrent requests have to be handled by some central server on the cloud. For example, during dinner time, delivery apps are used by millions of users that place orders and then track the delivery on a map. The central tracking system used by the take-out meal providers should be able to withstand such a high volume of requests. Creating and managing a server that can react to bursts of requests can be indeed challenging. Even worse is that in the off-peak hours, the server would be under-used. Hence it needs to elastically acquire and release computational resources to maximize resource utilization.

Fortunately, with a serverless architecture, such complicated server resource management can be fully delegated to the cloud service provider, such as AWS. All the developers need to care about is the code that enables location information to flow from the uploaders to the users. Naturally, people might question whether such a design can be reliable, robust, and profitable. To investigate the potential of serverless architecture, we come up with our objectives as follows.

2.2 Objectives

Our overall objective is to implement a tracking application and to mimic real-life situations with a high throughput of data. The application should show the most basic information that our clients are interested in (e.g. the progress of the delivery with location and sampling time) in a direct way, and also be able to handle concurrent updates of data with a serverless structure. To mimic the high throughput, the application should also be associated with an engine that can concurrently generate data at a large scale.

Therefore, the first main goal is to develop an interactive web application that supports querying for the current location of devices, in order to track and trace various services, such as a delivery. And the other one should be a simulation engine that generates data and supports pressure testing the tracking system.

2.2.1 Target 1 (Interactive Tracking Interface)

The objective is to display the progress of services that require location tracking, according to the query parameters entered by the users. They can search using unique `deviceID` (a real-life example would be the id of a location uploader, such as the unique identifier of a deliveryman) to check the status of a delivery, and stores could monitor the progress as well by querying with a `trackerID` (could be represented as an identifier for a group of deliverymen with multiple orders). With such a delivery tracking system, we also need to verify the effectiveness of this serverless architecture-based implementation.

2.2.2 Target 2 (Simulation Engine)

This target involves concurrently generating location data of devices, at a large scale, and stress testing the architecture. Using the simulation engine we could mimic the real-life situations with concurrent requests and data updates, and enabled us to pressure test the performance of this serverless architecture. We need to evaluate the performance statistics of such implementation as part of this objective.

2.3 Achievements

To be completed We managed to simulate up to 1,000 devices publishing their positions to the tracking system, while displaying their positions on the map on the user web application. We have also run multiple rounds of simulation and compared the performance of the tracking system under different throughput.

3 Design and Implementation

3.1 Overall Design

Our design includes three major components. For the Delivery Tracking Interface, we have a back-end to handle location uploads and store them. Also a front-end to display stored location information is implemented. The last component is a simulation engine used to pressure-test the back-end and gather performance statistics. Figure 1 shows the basic design idea.

3.2 Interactive Tracking Interface

3.2.1 User Upload Handler: AWS IoT Core

Being able to upload location data is one of the essential features of a delivery tracking system. Users will not be able to track any deliveries unless there is a device associated with each delivery that provides updates of its location at regular intervals or when checkpoints are reached.

To upload location data for a device, we first locate and retrieve its credentials locally (i.e. certificate, private/public key). These are then used to construct an MQTT connection to an AWS IoT Core endpoint. After the connection has been established, we can then publish the location data update to a specified MQTT topic on AWS IoT Core.

We have decided to use AWS IoT Core and MQTT as they were a perfect fit for our purposes. Each tracking device used for deliveries could be seen as an IoT thing so it was obvious to use AWS IoT Core for the location updates. On the other hand, MQTT is a lightweight messaging

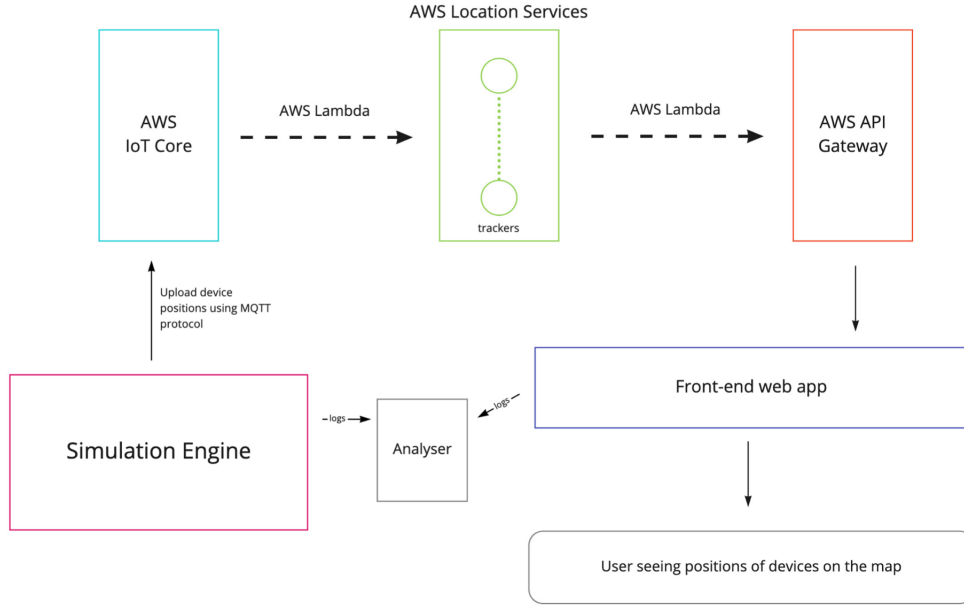


Figure 1: Overall Architecture of Design

protocol which makes it suitable for our project where the actions performed by the devices are limited.

Once the location data has been published to IoT Core, an AWS lambda function that is subscribed to that topic will be triggered. It will first assign a suitable AWS location services tracker for the device, after which the location update data will be sent to the tracker and each tracker contains location data for multiple devices. More information regarding the AWS location services trackers is given below. [5]

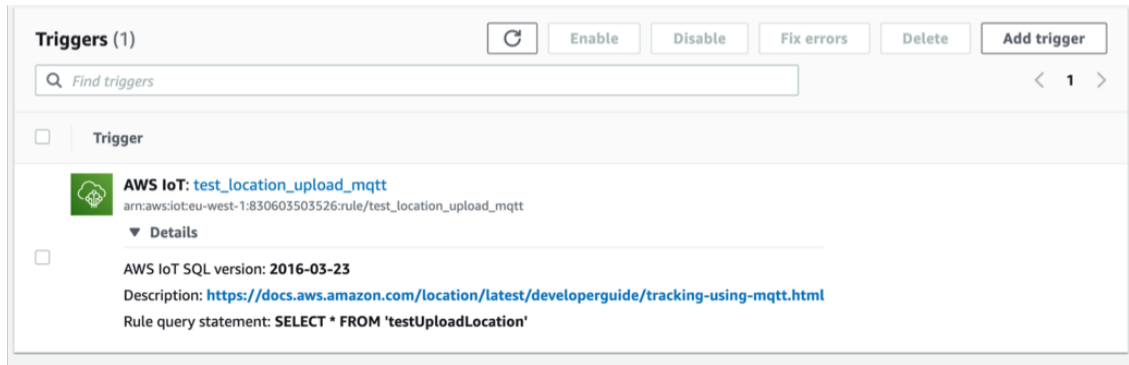


Figure 2: Trigger for an AWS Lambda [1] function

3.2.2 Data persistence: AWS Location Service Trackers

We use 10 (ten) Trackers from AWS Location Service to retain the uploaded location data. The trackers will receive device position updates from an AWS Lambda Function, which helps to balance the load of device updates. Each tracker can store the location data for multiple devices. The trackers implement time-based filtering on received updates, which means that device position

updates with sampled time differed by less than 30 seconds will be ignored. With time-based filtering, in a real-world scenario, the per-device update frequency will be less than 2 times per minute, saving cost by rejecting unnecessary updates. The trackers are queried by another AWS Lambda Function to retrieve the stored data upon request.

```

PS C:\Users\Annnn> aws location list-device-positions --tracker-name sim-iot-tracker1
{
  "Entries": [
    {
      "DeviceId": "AYZ-1056",
      "Position": [
        36.165993,
        -86.778815
      ],
      "SampleTime": "2021-12-29T03:13:24+00:00"
    },
    {
      "DeviceId": "AWS-1234",
      "Position": [
        -123.1187,
        50.2819
      ],
      "SampleTime": "2022-01-09T11:48:59+00:00"
    },
    {
      "DeviceId": "EGG-1263",
      "Position": [
        36.193302,
        -86.830646
      ],
      "SampleTime": "2022-01-09T12:40:05+00:00"
    }
  ]
}

```

Figure 3: Device positions stored in a tracker

We considered many potential AWS products as the persistence medium for storing device position updates. Before we start comparing different potential solutions, we should highlight the key characteristics of location data. Location data are lightweight, exchanged frequently and most of the time, only the most recent data matter. [6] [7] [8]

	Trackers	S3 Buckets	Timestream Database
Pros	Suitable for storing location data; Easily extensible to work with other products in AWS Location Services	Suitable for storing location data; The cost per GET request is low	Cost-effective for storing time series data
Cons	Slightly higher cost	Provides storing and retrieval of unstructured or semi-structured data only; All other functionalities must be implemented manually	Designed to be queried for analytical purposes; Frequent querying of the latest update can be expensive

We do encounter some challenges working with trackers. The trackers automatically reject device position updates if their sampled time is more than thirty days ago. The generated fake device positions data used in simulation usually have a very old timestamp. Hence, we need to add a fixed offset to them to be accepted by the trackers. We also must increment the offset for every round of simulation, otherwise, the trackers would not recognize these data as their sampled time is less recent than that stored in the trackers.

3.2.3 Data Fetching: AWS Lambda

This is the core part connecting the frontend to the backend. Once the users search for a device location, the application should fetch the corresponding data from the backend and display the retrieved data on the frontend. Here we use **AWS Lambda** for managing the underlying computational resources [1] and retrieving data from the backend..

The core idea of our design is to have a function that is lightweight but responds quickly to events. Furthermore, it should handle different query parameters for various use cases. For example, customers care more about the current location of delivery and how the delivery was going since dispatched. In this case, we should include the location history of devices, with different timestamps. However, stores, for example, care more about tracking a group of deliveries in real-time, so the response here should include the current location of a group of tracked devices, according to a specific "tracker" identifier.

For more details of the implementation, firstly we have an API Gateway as the trigger of this lambda function. This provides us with an endpoint of invoking the lambda function and forms a clear pipeline after receiving the event updates. In the lambda, we bind the location client with AWS Location Services, an underlying computing resource of this application. And once we received an event update with appropriate query parameters, e.g. *type* of the information and the *trackerName*, we would use the provided methods from the client to fetch data from the backend, e.g. the list of information of devices controlled by a tracker. Finally, we would return a response containing the status code of this query and the main body with fetched data.

The reason for this design is, firstly we are implementing an application with a serverless structure, we need to make the design as simple as possible. We have a powerful lambda for handling event updates, but the interface is not elegant enough to hide the details and provide a simple endpoint for HTTP Requests. So we choose to use **API Gateway** for constructing a pipeline of handling requests and event updates. The pipeline hides the core functionality of the lambda function well but preserves a clean interface for the front-end to fetch data. The pipeline also guarantees the safety of using lambda by adding appropriate authentications in separate stages.

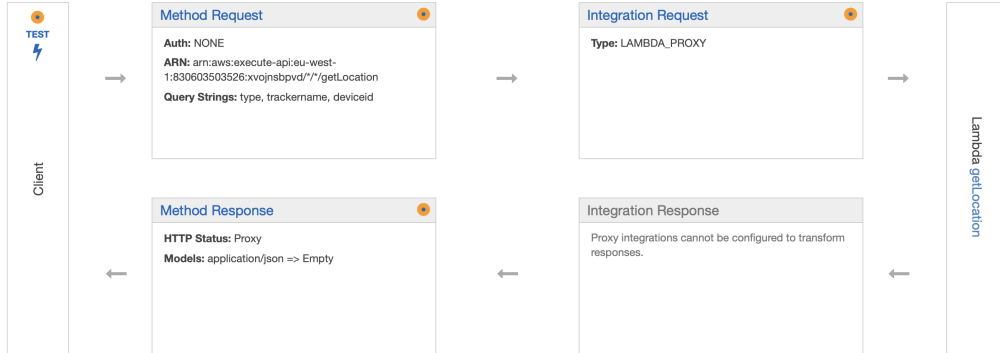


Figure 4: An Example of API Gateway [2] Pipeline

Indeed we need to consider the efficiency of the lambda function itself. If our lambda function is implemented incorrectly and does not preserve the serverless feature, e.g. fast response speed and low resource consumption, the effectiveness of this tracking system cannot be guaranteed. To prevent the underlying risk, we set the timeout metric for the lambda function to ensure execution within a short period. Also, we are using the location information of potential customers, so we should also consider data protection and safety measures. Here we attach an AWS IAM (Identity and Access Management) policy and manage authentications to ensure safety.

3.2.4 User Interface: React App

Due to the nature of our project, we had to work with and display location data that had to be kept up to date according to the tracked devices' current location, in a convenient way for the user.

A Document Object Model (DOM) is a programming interface that dictates the structure of web documents by creating a tree, containing objects and nodes. As seen in Figure 5, each branch in a DOM ends with a node, containing objects which can be modified using a scripting language, such as JavaScript, thus allowing the creating of dynamic web pages[9, 10].

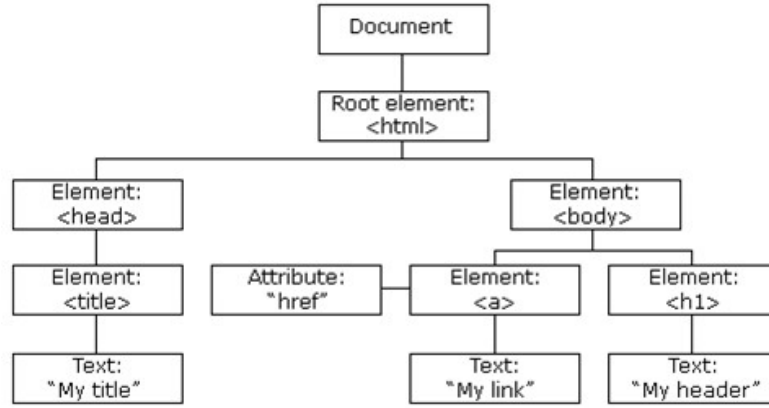


Figure 5: A DOM generated by a simple HTML application [3]

Directly modifying a DOM using a scripting language, in order to make a web page dynamic, is inefficient, tedious, and hard to scale. Thus, we have decided to build the web application using React, a free open-source library developed by Meta and distributed under the MIT license. React creates a lightweight, more efficient copy of the DOM using JavaScript, called virtual DOM (VDOM). When a part of the VDOM must be updated, React will only rerender the corresponding part of the VDOM, thus providing an improved performance and saving the programmer from having to do the tedious work of updating the DOM [11, 12].

React enabled us to easily display live location data and track devices on a map in real-time. In order to keep the data up to date, we used React's state functionality and made the VDOM update whenever the data inside the used state gets updated by the API calls to the created AWS API Gateways.

However, state updates are a double-edged sword. One issue that we faced multiple times was state update cycles, where a state change would trigger a re-render of a component that would trigger again that specific state update, thus resulting in an infinite cycle. We solved these issues by making sure we only update the states at key moments and making sure we have no unnecessary state updates.

To maintain a similar consistent design in the application and save time, we have decided to use the Material UI Core [13], a free open-source library distributed under the MIT license. For the mapping functionality, we used React Leaflet [14] (open-source library distributed under the Hippocratic License).

3.3 Simulation Engine

Let's first describe what the simulation engine is. It is a tool that simulates and tests the use of our serverless architecture. More precisely, it tests out the implementation of our tracking system.

It creates the location data and the devices, and also performs the publishing of the location updates. It can simulate a large number of concurrent devices. It has been mainly developed in Scala but also has a script in Python and uses a third-party data simulator in JavaScript.

3.3.1 Generating the necessary data

This is handled by the method `initialize_sim_and_devices_data`.

First, we need to generate the location data. For that purpose, we use the tool trip-simulator created by SharedStreets [15]. It generates simulated raw GPS telemetry, following a model that allows the simulation of a set of moving agents (car, scooter or bikes). It resolves the problem of privacy because it produces artificial data. It also adds noise to the data to be appropriate for implementations like ours that need to use data as realistic as possible. It outputs 1hz location probes, which we are going to call simulation files. Each probe or simulation file contains the generated data for a set of devices, and as we want to obtain a large number of devices, we concurrently run the trip-simulator tool to create several probes.

Given that this tool features a command-line interface, we call it from our Scala code and we use afterwards the files generated by it. In our case, we have decided to simulate cars because they move faster than scooters or bikes. This will help to notice position changes in the front-end. The chosen location for the simulation has been Nashville, in the United States, and we simulate two hours of movement. This gives us a large amount of data as the positions are updated each second. Finally, as the tool associates timestamps to the generated location updates, we choose that the simulation takes place approximately 10 days before the exact date when the simulation engine is run. This is essential as it allows us to reuse the generated data adding an offset. We will see this more in detail later.

As we mentioned earlier in the introduction of the project, each simulated device acts as an IoT Thing. Finding a way to automatically build the corresponding IoT Things on IoT Core has been a challenge during the development of the simulation engine. A key and a certificate have to be created for every new device (IoT Thing). To this effect, the simulation engine first creates a file containing the names of all the generated devices from the trip-simulator probes and then run a Python script that performs the task that correctly set up the IoT Thing. The script creates an IoT thing, sets up appropriate access rights for the IoT thing and writes received credentials into files. The script mainly makes use of APIs in AWS Python SDK.

3.3.2 Parsing the data, updating the offset

Parsing and transforming the data from the simulation files is indispensable to make the most of the generated data. Not all the information in a line of a probe file is necessary, and we create a list containing each entry of that file in a more appropriate format. Trackers in the Amazon Location Service operate with one of the following three position filtering methods: accuracy-based, distance-based and time-based filters. This is an implementation choice from Amazon that provides efficiency as the tracker ignore irrelevant updates.

Given that the main goal of the simulation engine is testing the performance and proving the correctness of our architecture, the trackers must accept all the messages we send. Therefore, because we want to preserve the realistic aspect of our data, we select the time-based filter: only one update per 30 seconds is stored for each unique device ID. In other terms, the timestamps included in two consecutive location updates for the same device have to be at least 30 seconds apart. We are talking about the timestamp that belongs to the location message (a notification containing the position and when that location was recorded), not the time of reception of the message by the tracker. This way, we parse and filter our simulation files so that our messages follow this rule.

We have not yet explained why we generate the simulated data with 10-day-old timestamps. That is for reusing data. As we already mentioned, each time we generate the simulation files we also have to set up the new IoT things. To avoid this, we reuse that data thanks to adding an offset to the timestamps. We keep track in a file of the current offset, and each time we run the sending part of the simulation engine we update the offset by intervals of a bit more than two hours (the simulated time) until the threshold of 10 days is reached: we do not want to send updates happening in the future.

Coming back to the parsing theme, we have also implemented two other methods that batch or group the list of messages depending on how (and where) they will be sent: to IoT Core or directly to a tracker. We will continue explaining these two sending methods in the next subsection.

3.3.3 Sending the location updates

So far, we have been mentioning that we send the location updates to IoT Core. However, before carrying out high-throughput sending simulations, we have implemented two simple methods, `trackers_test` and `iot_test` that upload the data of a device directly to a tracker or to an IoT Thing, respectively. Using the first option, specifying a certain tracker and device, we verify that we can correctly send and retrieve information about the position of a single device directly to a tracker. Similarly, with `iot_test`, we can verify the publication to IoT Core.

Nevertheless, our goal for this simulation engine is to observe how our serverless architecture handles high-throughput location updates. For that purpose, we need to send messages from several devices concurrently. Our first approach, even before we wrote the complete implementation of the parser methods that we explained before, was to concurrently send updates of live locations, i.e. messages with current timestamps from different devices at the same time, thanks to an array of concurrent tasks. In that way, we were able to check that the front-end could handle immediate updates. As the 30-seconds filter was applied, the throughput was very low, as a constant amount of devices (limited by the number of threads) had to wait to send subsequent updates. Thus, we preserved the array of concurrent tasks, but we implemented the parser techniques and operated with less recent timestamps that allowed us to send updates with a much higher frequency. The 30-seconds filter only applies to the timestamps (the location information itself) of two compared messages and has nothing to do with the frequency at which that updates are received.

It should be mentioned that non-current timestamps are successfully accepted by the trackers, and if a message contains a timestamp that becomes the most recent one for a certain device this update is visible on the map. Nevertheless, a tracker, concerning a single device, has to receive timestamps in increasing order: if it receives a timestamp older than another already stored for that device, it will throw an error. Using adequate concurrency implementations and a valid offset guarantee a correct order of the timestamps. At least locally, because we encountered a problem due to how the lambda function works. But we will explain this later, considering that it also applies to our next concurrency model and also can be interpreted as an evaluation of the architecture.

This concurrent implementation was very limited in throughput. We created an array of tasks, and each task consisted of a method publishing messages from an assigned simulation (probe) file. This was not optimal because the number of concurrent tasks was limited to the number of threads. And, given that each simulation file contains messages from several devices, a task i , say, a task assigned to a probe file i , first established a connection to a specific IoT thing, sent all the messages for that device, and then continued with the following device. This was not what we searched for as we were just achieving a reduced amount of real concurrent devices, even though the throughput of messages was sufficiently high.

Thus, we implemented a more advanced concurrency model. We followed the Akka Actor model [16]. We have defined two types of actors: an `IoTManager` and an `IoTThing`, and they communicate with a defined set of messages (called operations) based on requests and acknowledgements. In

our `multiple_actor_iot_test` method, we create a unique `IoTManager` and we initialize all the `IoTThing` actors, each one associated with a device. This way, all the IoT MQTT connections with AWS are set before starting the procedure of sending the location messages, then all the devices will be ready to send the updates immediately. The `IoTManager` keeps track of the active actors and requests and acknowledges the location updates of every device to guarantee a correct message publishing order. The manager also handles the stop of the rest of the actors, including itself and the `ActorSystem`.

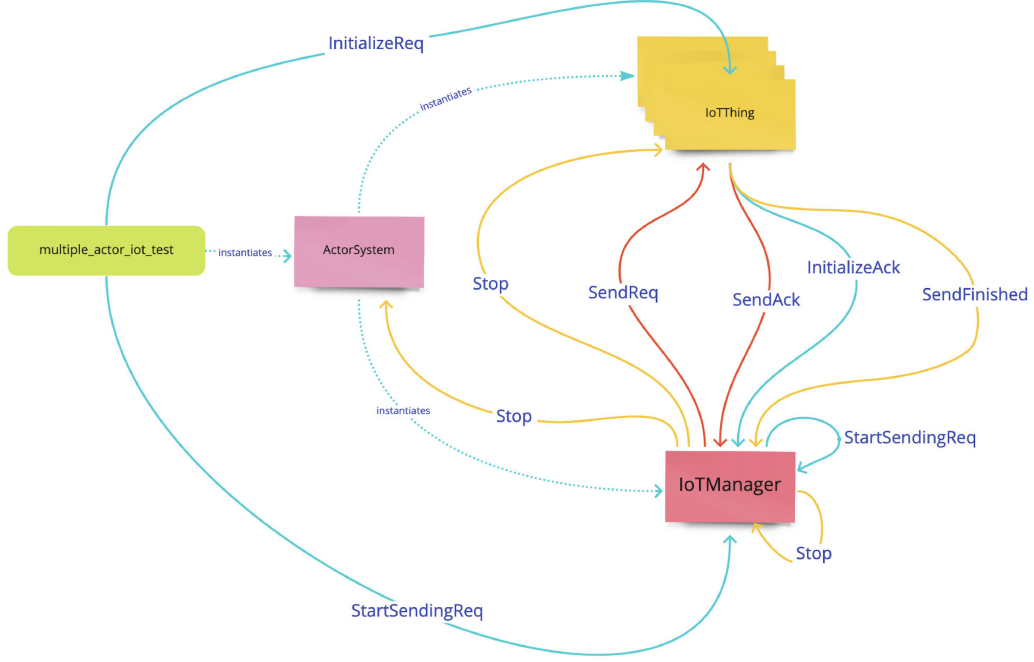


Figure 6: Concept map describing our Akka Actor model

3.4 Analyse Simulation Results

3.4.1 The Simulation Log Analyser

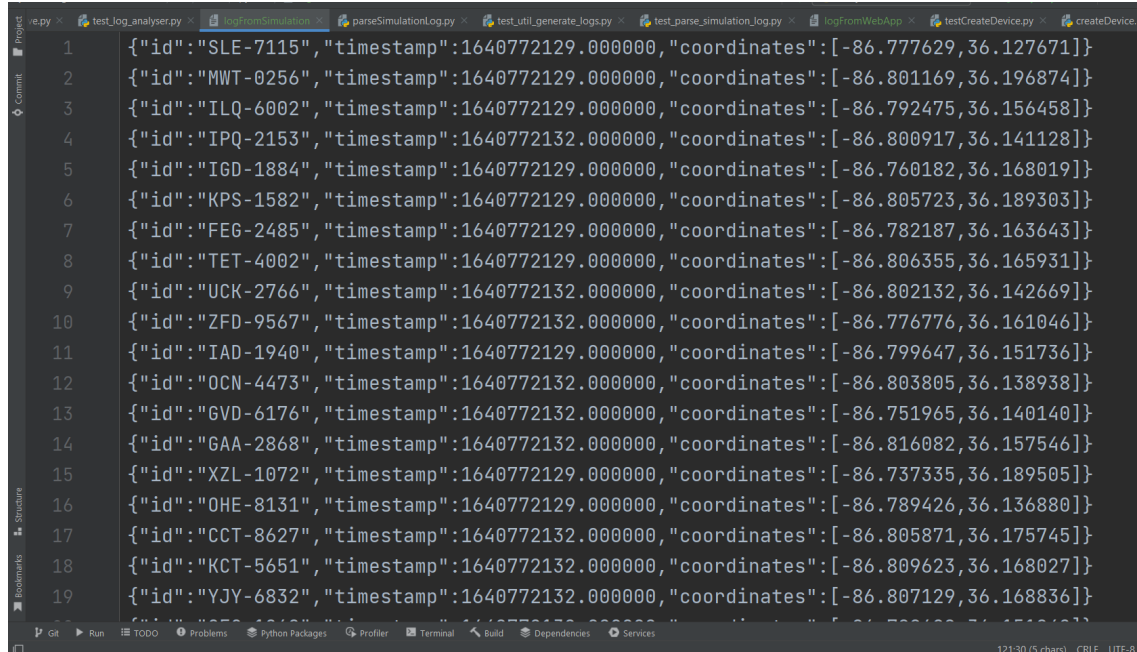
The log analyser analyses logs produced by the simulation engine during one round of simulation, as well as the log produced by the user front-end that retrieves device position data from the tracking system. The simulation engine produces log entries that record the timestamp of the event publishing a device’s position data to the AWS IoT core.

The user front-end produces a similar log, with the timestamp representing received time instead.

The analyser first parses the log and categorizes the log entries by its device ID, which is unique. For each device ID, the tuples (timestamp, position), are stored in ascending order of the timestamp. This sorted order will aid us in matching an entry in the next stage.

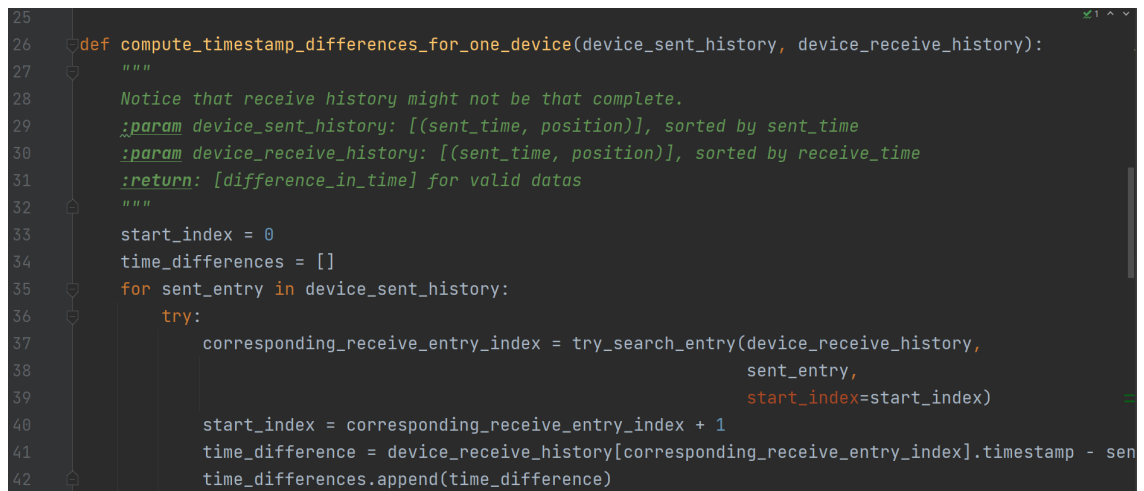
For each device and for each entry in the “send log”, the analyser searches for a matching entry in the “receive log”. Two positions match if they belong to the same device ID and are sufficiently close to each other. In matching the next entry in the send log, we only start searching in the receive log one entry after the previously matched one. This helps to reduce matching time. Whenever a matched pair is found, the timestamp difference is recorded.

After we find the timestamp differences for every device, we can look at their distribution, which represents a relative measure of overall latency in the tracking system. When we perform multiple



```
1 {"id": "SLE-7115", "timestamp": 1640772129.000000, "coordinates": [-86.777629, 36.127671]}
2 {"id": "MWT-0256", "timestamp": 1640772129.000000, "coordinates": [-86.801169, 36.196874]}
3 {"id": "ILQ-6002", "timestamp": 1640772129.000000, "coordinates": [-86.792475, 36.156458]}
4 {"id": "IPQ-2153", "timestamp": 1640772132.000000, "coordinates": [-86.800917, 36.141128]}
5 {"id": "IGD-1884", "timestamp": 1640772129.000000, "coordinates": [-86.760182, 36.168019]}
6 {"id": "KPS-1582", "timestamp": 1640772129.000000, "coordinates": [-86.805723, 36.189303]}
7 {"id": "FEG-2485", "timestamp": 1640772129.000000, "coordinates": [-86.782187, 36.163643]}
8 {"id": "TET-4002", "timestamp": 1640772129.000000, "coordinates": [-86.806355, 36.165931]}
9 {"id": "UCK-2766", "timestamp": 1640772132.000000, "coordinates": [-86.802132, 36.142669]}
10 {"id": "ZFD-9567", "timestamp": 1640772132.000000, "coordinates": [-86.776776, 36.161046]}
11 {"id": "IAD-1940", "timestamp": 1640772129.000000, "coordinates": [-86.799647, 36.151736]}
12 {"id": "OCN-4473", "timestamp": 1640772132.000000, "coordinates": [-86.803805, 36.138938]}
13 {"id": "GVD-6176", "timestamp": 1640772132.000000, "coordinates": [-86.751965, 36.140140]}
14 {"id": "GAA-2868", "timestamp": 1640772132.000000, "coordinates": [-86.816082, 36.157546]}
15 {"id": "XZL-1072", "timestamp": 1640772129.000000, "coordinates": [-86.737335, 36.189505]}
16 {"id": "OHE-8131", "timestamp": 1640772129.000000, "coordinates": [-86.789426, 36.136880]}
17 {"id": "CCT-8627", "timestamp": 1640772132.000000, "coordinates": [-86.805871, 36.175745]}
18 {"id": "KCT-5651", "timestamp": 1640772132.000000, "coordinates": [-86.809623, 36.168027]}
19 {"id": "YJY-6832", "timestamp": 1640772132.000000, "coordinates": [-86.807129, 36.168836]}
```

Figure 7: Sample simulation log



```
25
26 def compute_timestamp_differences_for_one_device(device_sent_history, device_receive_history):
27     """
28     Notice that receive history might not be that complete.
29     :param device_sent_history: [(sent_time, position)], sorted by sent_time
30     :param device_receive_history: [(sent_time, position)], sorted by receive_time
31     :return: [difference_in_time] for valid datas
32     """
33     start_index = 0
34     time_differences = []
35     for sent_entry in device_sent_history:
36         try:
37             corresponding_receive_entry_index = try_search_entry(device_receive_history,
38                                                                     sent_entry,
39                                                                     start_index=start_index)
40             start_index = corresponding_receive_entry_index + 1
41             time_difference = device_receive_history[corresponding_receive_entry_index].timestamp - sent_entry.timestamp
42             time_differences.append(time_difference)
```

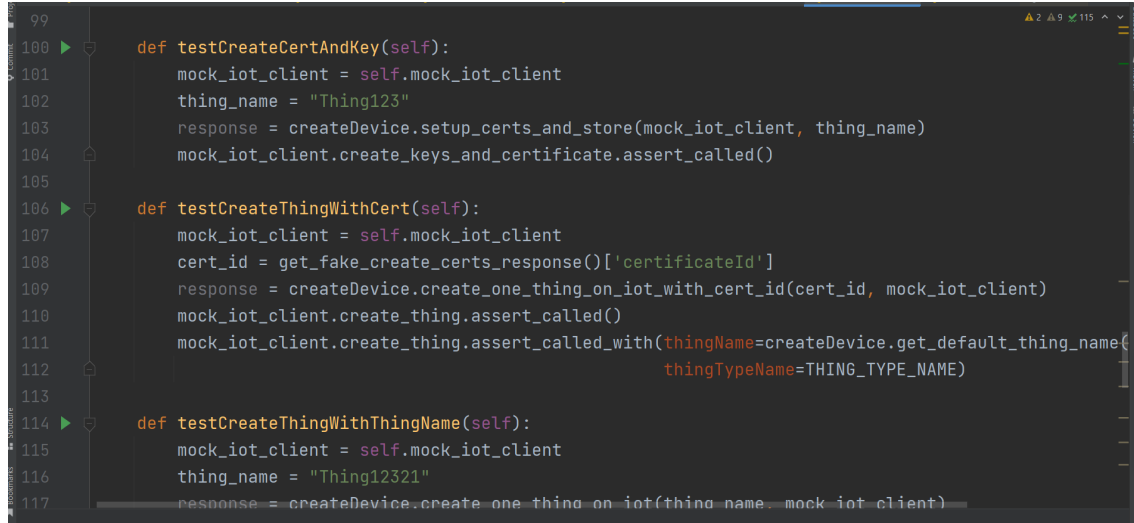
Figure 8: Finding the timestamp difference in matched entries

rounds of simulation with increasing number of simulate devices, we can observe if there is a significant increase in the mean of the distribution, which would signal that some bottleneck is encountered. We can also look at the distribution of matched positions per device, which represents the reliability of the tracking system.

4 Evaluation

4.1 Unit Testing For Each Component

The device creation script is unit-tested with mock AWS client to ensure correct behaviour.



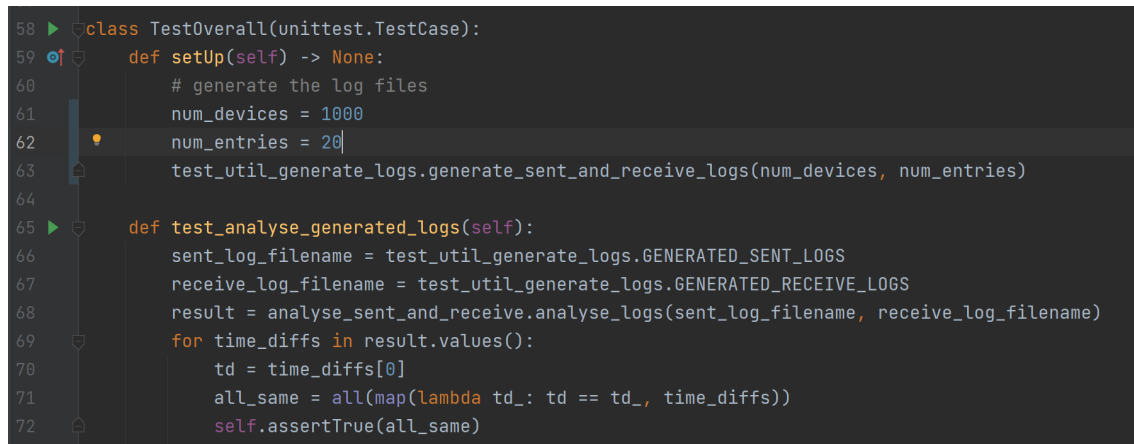
```

99
100 def testCreateCertAndKey(self):
101     mock_iot_client = self.mock_iot_client
102     thing_name = "Thing123"
103     response = createDevice.setup_certs_and_store(mock_iot_client, thing_name)
104     mock_iot_client.create_keys_and_certificate.assert_called()
105
106 def testCreateThingWithCert(self):
107     mock_iot_client = self.mock_iot_client
108     cert_id = get_fake_create_certs_response()['certificateId']
109     response = createDevice.create_one_thing_on_iot_with_cert_id(cert_id, mock_iot_client)
110     mock_iot_client.create_thing.assert_called()
111     mock_iot_client.create_thing.assert_called_with(thingName=createDevice.get_default_thing_name(
112         thingTypeName=THING_TYPE_NAME)
113
114 def testCreateThingWithThingName(self):
115     mock_iot_client = self.mock_iot_client
116     thing_name = "Thing12321"
117     response = createDevice.create_one_thing_on_iot(thing_name, mock_iot_client)

```

Figure 9: Mock test for IoT Device Creation Script

The components of the analyser are unit-tested. We have tests for the underlying data structure as well as integration tests for the analyser. We also generate fake simulation logs and receive logs that contain about 20,000 entries to test for the efficiency of the analyser.



```

58 class TestOverall(unittest.TestCase):
59     def setUp(self) -> None:
60         # generate the log files
61         num_devices = 1000
62         num_entries = 20
63         test_util_generate_logs.generate_sent_and_receive_logs(num_devices, num_entries)
64
65     def test_analyse_generated_logs(self):
66         sent_log_filename = test_util_generate_logs.GENERATED_SENT_LOGS
67         receive_log_filename = test_util_generate_logs.GENERATED_RECEIVE_LOGS
68         result = analyse_sent_and_receive.analyse_logs(sent_log_filename, receive_log_filename)
69         for time_diffs in result.values():
70             td = time_diffs[0]
71             all_same = all(map(lambda td_: td == td_, time_diffs))
72             self.assertTrue(all_same)

```

Figure 10: Pressure test of the analyser

The AWS Lambda Functions are also unit-tested locally.

We use the MQTT test client provided by AWS IoT, where sample location data was used, to make sure that the IoT connections were successful and that the devices were able to perform publishing to MQTT topics.

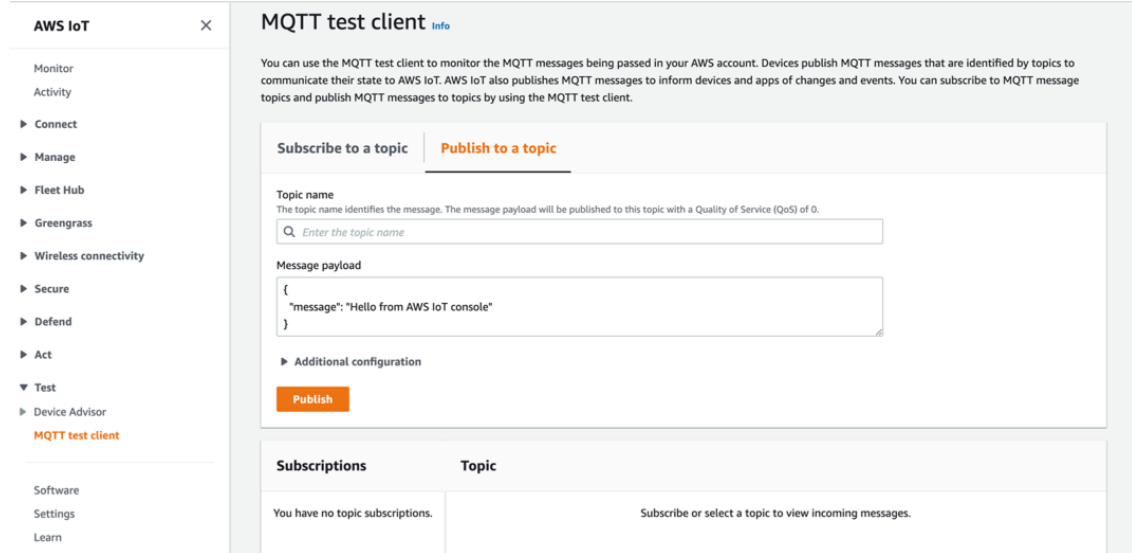


Figure 11: AWS MQTT test client

4.2 Integration Test

To evaluate the performance of the whole tracking system, we use our simulation engine to simulate up to 1000 unique devices uploading their device position data concurrently and use the user frontend to observe real-time updates of the positions of markers displayed. After the simulation has finished, we use the analyser to make sense of the log produced during the simulation.

4.3 CI/CD Pipeline

Our pipeline ensures there are no errors and warnings and immediately deploys the web application to Heroku. This enabled us to easily monitor the progress of our project and test new implemented features by sharing the application with the team, the supervisor and other volunteers who provided us with feedback.

4.4 Simulation engine

With the actor model, we have successfully performed a simulation of 1000 concurrent devices. As we expected, high-throughput is not a problem for our architecture and it handles well that amount of traffic. This confirms that AWS is the correct platform to build up this type of product, as we mentioned at the beginning of the report. However, we encountered a common error after testing both implementations of sending location updates (the one with the array of tasks and the another with the actors): when the lambda that gets from IoTCore and publishes to the trackers scales up, it fetches some messages in a reordered way, causing that the trackers reject some updates because the timestamps are no longer assured to be handled in the correct order.

Observing the log file and the local output of our simulation engine, we can confirm that the messages are sent in the correct order (this is guaranteed because of how the actors have been implemented, and also because of the offset, which assures that no overlapping of timestamps occurs between two simulations). We can observe the error because of the lambda on CloudWatch

logs: `{'Code': 'ValidationError', 'Message': 'timestamps older than already stored not allowed'}`. We can also perceive it in CloudWatch metrics.

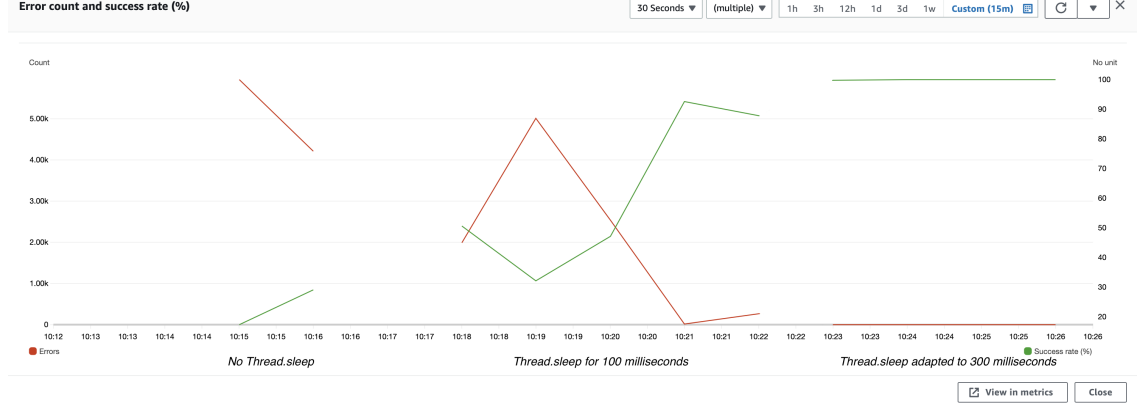


Figure 12: Lambda metrics for a 30-device simulation with the actor model

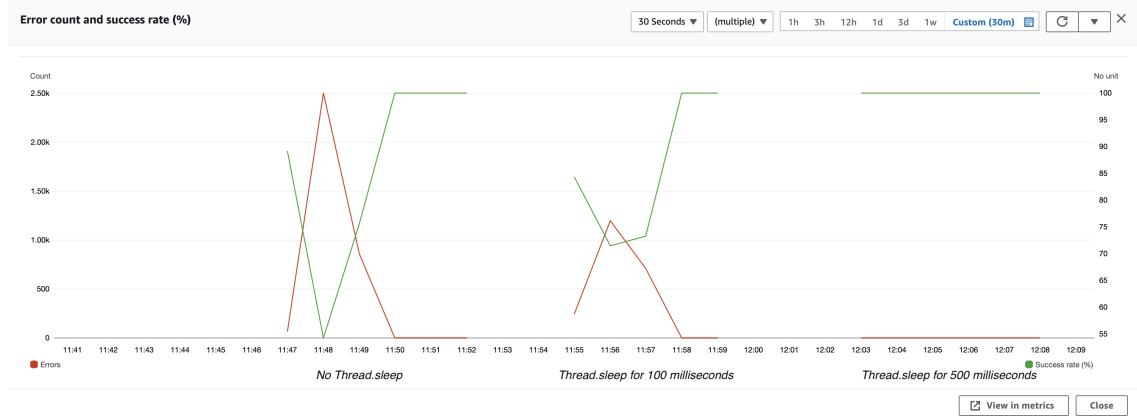


Figure 13: Lambda metrics for a 30-device simulation with the task array model (15 sim. files and 2 devices per file)

To solve this problem, we have added in the `format_and_publish_msg` method, after the get that waits for the Future publication of the location update to complete, a `Thread.sleep` that forces a delay between sending two messages from potentially the same device. This solution completely reduces the errors. The number of milliseconds that the thread has to sleep is determined by the number of devices in the actor model: the more devices the faster the thread can be reactivated, because more time will be gone between two messages of the same device as usually the entire set of things has to be traversed before a device is again sending a new location update. In the case of the former implementation with the array of tasks, the sleeping time has to be constant and higher, as all the messages from the same device are sent one after the other before passing to the next device in the corresponding simulation file.

Finally, we can conclude that, adopting a correct approach of the concurrency model and adapting to the behaviour of the lambda, our architecture successfully pass our evaluations powered by the simulation engine and shows that AWS is a suitable platform to develop high-throughput serverless location products.

4.5 User Feedback

Our supervisor is satisfied with our tracking system, as it demonstrates the ability of a serverless cloud architecture in the scenario of tracking devices.

5 Ethical Considerations

The goal of the project is to create an application that will help in benchmarking serverless architecture for tracking the location of multiple devices, potentially helping people, such as application developers, into deciding whether a serverless tracking system is more profitable compared to a traditional, serverful, location tracking system.

Thus, the project involves location data collection and processing, which is securely stored and managed on the AWS servers, and we ensured the permissions are set accordingly in order to restrict unauthorised access.

For our simulations, we have created the IoT things such that every IoT thing had the authority to publish and subscribe to any MQTT topic. While this has no impact on our simulation due to the fact that the devices are not publishing real-time location data, it would not be a good practice to adopt in the scenario of real users and devices. Then it would be possible for unrelated devices to acquire each other's location data which could lead to security problems. In a realistic scenario, it would be sensible to assign a unique order number for each delivery and a unique MQTT topic associated with it. Then we would be able to create the IoT things and policies such that each device only has access to MQTT topics that are relevant to itself.

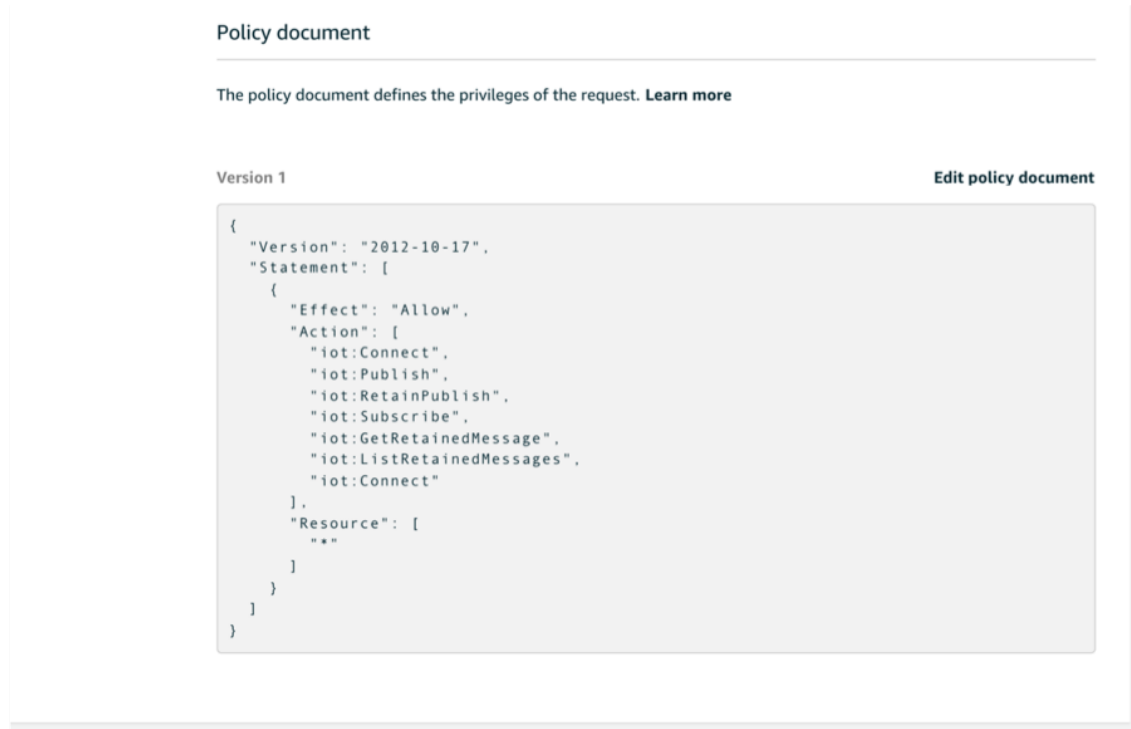


Figure 14: An IoT thing policy

In our case, we are using generated realistic location data, however, the project can easily be adapted and misused by ill-intentioned people, for example, in tracking unaware victims.

Although there are no intentions for the project to be used in a military context, it is possible that a similar architecture can be created to track military machines such as drones.

References

- [1] MUSGRAVE D. Serverless Computing - AWS Lambda - Amazon Web Services. EUROPA EDITIONS; 2022. Accessed: 2022-01-10. Available from: <https://aws.amazon.com/lambda/>.
- [2] Amazon API Gateway | API Management | Amazon Web Services. X.400 API Association; 1989. Accessed: 2022-01-10. Available from: <https://aws.amazon.com/api-gateway/>.
- [3] The HTML DOM (Document Object Model). W3Schools;. Accessed: 2022-01-09. Available from: https://www.w3schools.com/js/js_htmlDOM.asp.
- [4] Cloud Services - Amazon Web Services (AWS);. Accessed: 2022-01-10. Available from: <https://aws.amazon.com/>.
- [5] NEMETI F, PAULETTO G, DUAY D, COMTESSE X. IoT: l’emancipation des objets. Editions G. d’Encre.; 2017. Accessed: 2022-01-10. Available from: <https://docs.aws.amazon.com/iot/latest/developerguide/mqtt.html>.
- [6] Amazon S3 Simple Storage Service Pricing - Amazon Web Services. Strand Street Press; 2002. Accessed: 2022-01-09. Available from: <https://aws.amazon.com/s3/pricing/>.
- [7] Amazon Location Service Pricing - Amazon Web Services. Strand Street Press; 2002. Accessed: 2022-01-09. Available from: <https://aws.amazon.com/location/pricing/>.
- [8] Amazon Timestream Pricing - Amazon Web Services. Strand Street Press; 2002. Accessed: 2022-01-09. Available from: <https://aws.amazon.com/timestream/pricing/>.
- [9] Document Object Model. Wikipedia;. Accessed: 2022-01-09. Available from: https://en.wikipedia.org/wiki/Document_Object_Model.
- [10] Introduction to the DOM. Mozilla;. Accessed: 2022-01-09. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.
- [11] Virtual DOM and Internals. React;. Accessed: 2022-01-09. Available from: <https://reactjs.org/docs/faq-internals.html>.
- [12] Virtual DOM. Wikipedia;. Accessed: 2022-01-09. Available from: https://en.wikipedia.org/wiki/Virtual_DOM.
- [13] MUI Core. Material UI;. Accessed: 2022-01-09. Available from: <https://mui.com/core/>.
- [14] LE CAM P. React Leaflet. React Leaflet;. Accessed: 2022-01-09. Available from: <https://react-leaflet.js.org/>.
- [15] trip-simulator, generator of simulated raw GPS telemetry. SharedStreets; 2019. Accessed: 2022-01-09. Available from: <https://github.com/sharedstreets/trip-simulator>.
- [16] Akka 2.6.18 API. Lightbend; 2021. Accessed: 2022-01-10. Available from: <https://doc.akka.io/docs/akka/current/typed/actors.html>.