



# **Lab 1 Report**

50.046 CCIoT  
SUTD 2022 ISTD

*Group 3*

Han Xing Yi	( 1004330 )
Ang Song Gee	( 1004589 )
James Raphael Tiovalen	( 1004555 )
Velusamy Sathiakumar Ragul Balaji	( 1004101 )

# 1. Object Recognition with ESP32 CAM and AWS Services

## 1.1 Issues During Setup

### 1.1.1 “Could not open port” issue

When trying to upload the code into the ESP32 device, we faced an error where the current port had been set incorrectly.

```
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/songgee/.platformio/packages/tool-esptoolpy/esptool.py", line 4582, in <module>
    _main()
  File "/home/songgee/.platformio/packages/tool-esptoolpy/esptool.py", line 4575, in _main
    main()
  File "/home/songgee/.platformio/packages/tool-esptoolpy/esptool.py", line 4074, in main
    esp = esp or get_default_connected_device(ser_list, port=args.port, connect_attempts=args.connect_attempts,
  File "/home/songgee/.platformio/packages/tool-esptoolpy/esptool.py", line 120, in get_default_connected_device
    _esp = chip_class(each_port, initial_baud, trace)
  File "/home/songgee/.platformio/packages/tool-esptoolpy/esptool.py", line 313, in __init__
    self._port = serial.serial_for_url(port)
  File "/home/songgee/.platformio/penv/lib/python3.9/site-packages/serial/__init__.py", line 90, in serial_for_url
    instance.open()
  File "/home/songgee/.platformio/penv/lib/python3.9/site-packages/serial/serialposix.py", line 325, in open
    raise SerialException(msg.errno, "could not open port {}: {}".format(self._port, msg))
serial.serialutil.SerialException: [Errno 2] could not open port COM5: [Errno 2] No such file or directory: 'COM5'
*** [upload] Error 1
```

We rectified this error by setting the `upload_port` variable to `/dev/ttyUSB0` instead of `'COM5'`, within the `platformio.ini` file in the project's base directory.

```
upload_port = /dev/ttyUSB0
```

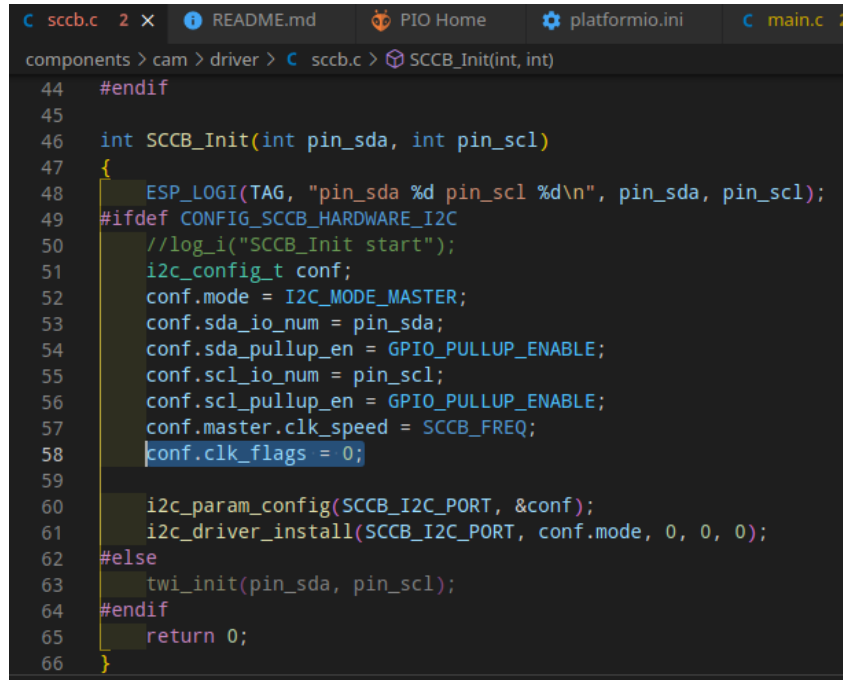
### 1.1.2 'i2c\_set\_pin' issue

When building and uploading the source code into the ESP32 Device, we received the error of “scl and sda gpio numbers are the same”.

```
I (3449) wifi:security: WPA2-PSK, phy: bgn, rssi: -51
I (3449) wifi:pm start, type: 1

I (3449) wifi:AP's beacon interval = 102400 us, DTIM period = 2
W (3479) wifi:<ba-add>idx:0 (ifx:0, 9e:7c:66:79:f1:69), tid:0, ssn:0, winSize:64
%[0;32mI (4069) esp_netif_handlers: sta ip: 192.168.195.219, mask: 255.255.255.0, gw: 192.168.195.145%[0m
%[0;31mE (4389) i2c: i2c_set_pin(828): scl and sda gpio numbers are the same%[0m
%[0;31mE (5389) i2c: i2c_set_pin(828): scl and sda gpio numbers are the same%[0m
%[0;31mE (6389) i2c: i2c_set_pin(828): scl and sda gpio numbers are the same%[0m
```

Upon further investigation (<https://github.com/espressif/esp-idf/issues/6293>), by adding the line `'conf.clk_flags = 0;'` into the `SCCB_Init` function in `components/cam/driver/sccb.c` file, this issue was resolved.



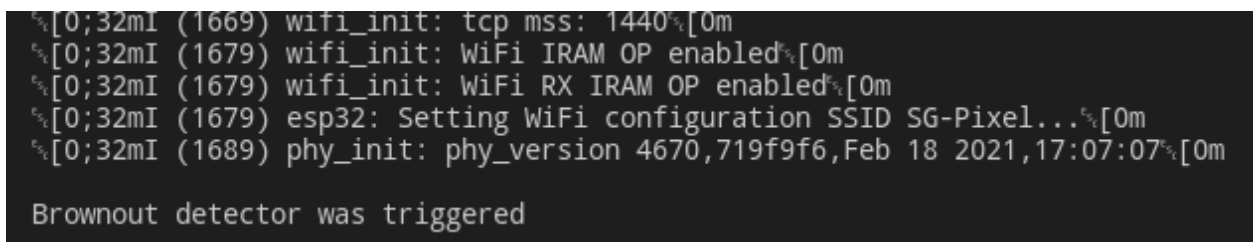
```

44 #endif
45
46 int SCCB_Init(int pin_sda, int pin_scl)
47 {
48     ESP_LOGI(TAG, "pin_sda %d pin_scl %d\n", pin_sda, pin_scl);
49 #ifdef CONFIG_SCCB_HARDWARE_I2C
50     //log_i("SCCB_Init start");
51     i2c_config_t conf;
52     conf.mode = I2C_MODE_MASTER;
53     conf.sda_io_num = pin_sda;
54     conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
55     conf.scl_io_num = pin_scl;
56     conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
57     conf.master.clk_speed = SCCB_FREQ;
58     conf.clk_flags = 0;
59
60     i2c_param_config(SCCB_I2C_PORT, &conf);
61     i2c_driver_install(SCCB_I2C_PORT, conf.mode, 0, 0, 0);
62 #else
63     twi_init(pin_sda, pin_scl);
64 #endif
65     return 0;
66 }

```

### 1.1.3 Brownout Detector Issue

When using the USB power supplied from the USB to I2C cable to power the ESP32 directly from our laptop power supply, we obtained a “Brownout detector was triggered” error in the Serial Monitor logs. This caused the device to continuously reboot due to the insufficient voltage supplied. To resolve this issue, we supplied power to the ESP32 using an external power supply.



```

[0;32mI (1669) wifi_init: tcp mss: 1440[0m
[0;32mI (1679) wifi_init: WiFi IRAM OP enabled[0m
[0;32mI (1679) wifi_init: WiFi RX IRAM OP enabled[0m
[0;32mI (1679) esp32: Setting WiFi configuration SSID SG-Pixel...[0m
[0;32mI (1689) phy_init: phy_version 4670,719f9f6, Feb 18 2021,17:07:07[0m

Brownout detector was triggered

```

### 1.1.4 Issue with sdkconfig file

When running the code, we received a ‘mbedtls\_net\_connect returned -0x52’ error when trying to connect to AWS. Before this, we had already tested our AWS IoT Thing’s Device Data Endpoint using Postman, thus we could isolate this error away from any AWS issues.

```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL
W (3739) wifi:<ba-add>idx:0 (ifx:0, 9e:7c:66:79:f1:69), tid:0, ssn:0, callb
I (3799) wifi:AP's beacon interval = 102400 us, DTIM period = 2
[0;32mI (4569) esp_netif_handlers: sta ip: 192.168.195.219, mask: 255.255.
145[0m
[0;32mI (4569) esp32: Connecting to AWS...[0m
[0;31mE (5109) aws_iot: failed! mbedtls_net_connect returned -0x52[0m
[0;31mE (5109) esp32: Error(-23) connecting to :8883[0m
[0;31mE (6649) aws_iot: failed! mbedtls_net_connect returned -0x52[0m
[0;31mE (6649) esp32: Error(-23) connecting to :8883[0m
[0;31mE (8179) aws_iot: failed! mbedtls_net_connect returned -0x52[0m
[0;31mE (8179) esp32: Error(-23) connecting to :8883[0m
[0;31mE (9709) aws_iot: failed! mbedtls_net_connect returned -0x52[0m
[0;31mE (9709) esp32: Error(-23) connecting to :8883[0m

```

Upon further investigation, when searching for other values of 'CONFIG\_AWS\_IOT\_MQTT\_HOST' in our project folder, we realized that although we had already set the value of 'CONFIG\_AWS\_IOT\_MQTT\_HOST' in our *sdkconfig* file, these changes had not been propagated to other files such as *sdkconfig.cmake*.

```

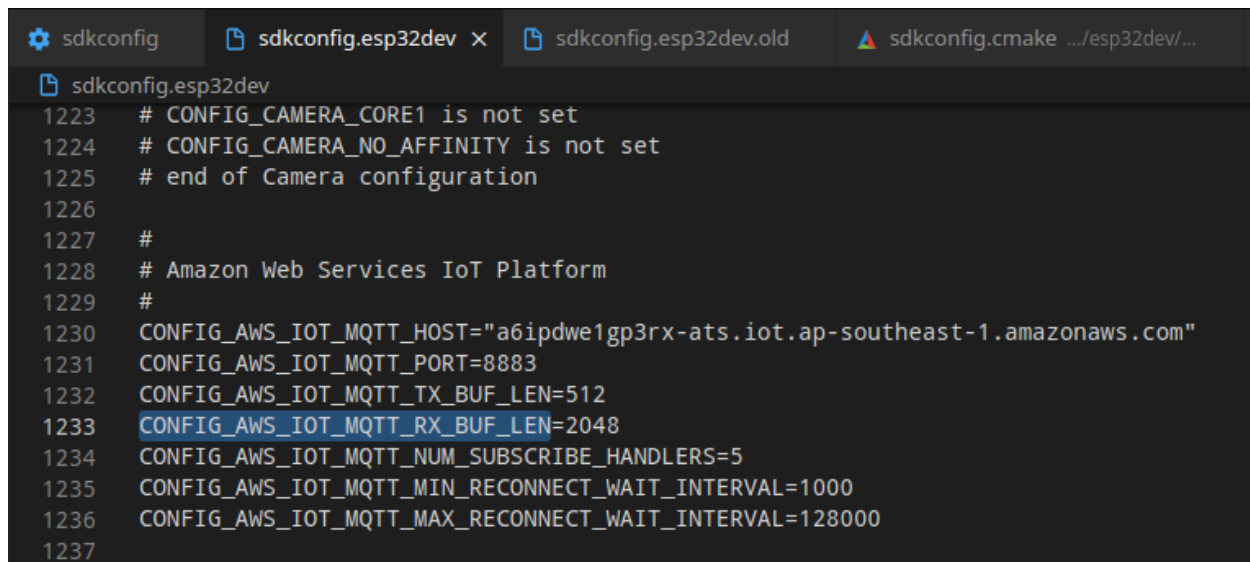
SEARCH
CONFIG_AWS_IOT_MQTT_HOST
Replace
10 results in 8 files - Open in editor
v sdkconfig 1
  CONFIG_AWS_IOT_MQTT_HOST="a6ipdwe1gp3rx-ats.iot.ap-southeast-1.amazonaws.com"
v sdkconfig.esp32dev 1
  CONFIG_AWS_IOT_MQTT_HOST=""
v sdkconfig.esp32dev.old 1
  CONFIG_AWS_IOT_MQTT_HOST=""
v sdkconfig.cmake .pio/build/esp32dev/config 2
  set(CONFIG_AWS_IOT_MQTT_HOST "")
  CONFIG_CAMERA_NO_AFFINITY;CONFIG_AWS_IOT_MQTT_HOST;CONFIG_AWS_IOT_MQTT_PORT;CONF...
v sdkconfig.h .pio/build/esp32dev/config 1
  #define CONFIG_AWS_IOT_MQTT_HOST 1
v sdkconfig.cmake build/config 2
  set(CONFIG_AWS_IOT_MQTT_HOST "")
  CONFIG_CAMERA_NO_AFFINITY;CONFIG_AWS_IOT_MQTT_HOST;CONFIG_AWS_IOT_MQTT_PORT;CONF...
v sdkconfig.h build/config 1
  #define CONFIG_AWS_IOT_MQTT_HOST 1
v aws_iot_config.h components/esp-aws-iot/port/include 1
  AWS_IOT_MQTT_HOST CONFIG_AWS_IOT_MQTT_HOST ///< Customer specific MQTT HOST. The ...

```

After manually replacing the values directly in the files listed above and testing, we found that we only needed to insert our endpoint link in the *sdkconfig.esp32dev* file for our device to connect to AWS successfully.

As a continuation of the above, we also faced an issue where the code of the device triggered an abort right after trying to fetch the s3 image URL from the MQTT 'esp32/sub/url' topic. The error code received was -32, which we found out to be MQTT\_RX\_BUF\_LEN\_EXCEEDED. Upon further investigation and debugging, we found that the main issue was that the device's MQTT receive buffer length was configured to be 512 in the *sdkconfig.esp32dev* file, while being configured as 1536 in the *sdkconfig* file. The *sdkconfig* file configurations were not being trickled down to the other files where the same configurations are being set.

We found the length of our s3 URL to be around 1200, which meant that our device had not been configured properly to accept such a long message. This confirmed our suspicions, and we resolved this issue by setting the 'CONFIG\_AWS\_IOT\_MQTT\_RX\_BUF\_LEN' config in *sdkconfig.esp32dev* and other relevant files to 2048, to accommodate the length of the message.



```
1223 # CONFIG_CAMERA_CORE1 is not set
1224 # CONFIG_CAMERA_NO_AFFINITY is not set
1225 # end of Camera configuration
1226
1227 #
1228 # Amazon Web Services IoT Platform
1229 #
1230 CONFIG_AWS_IOT_MQTT_HOST="a6ipdwe1gp3rx-ats.iot.ap-southeast-1.amazonaws.com"
1231 CONFIG_AWS_IOT_MQTT_PORT=8883
1232 CONFIG_AWS_IOT_MQTT_TX_BUF_LEN=512
1233 CONFIG_AWS_IOT_MQTT_RX_BUF_LEN=2048
1234 CONFIG_AWS_IOT_MQTT_NUM_SUBSCRIBE_HANDLERS=5
1235 CONFIG_AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL=1000
1236 CONFIG_AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL=128000
1237
```

### 1.1.5 Issue with AWS IoT Policy

When running the source code on the device, we faced an issue where the device was not able to receive the object labels of the captured image from the 'esp32/sub/data' MQTT topic. Instead, only an empty string was received. We managed to trace this error to the Esp32Policy under IoT Core Policy.

```

c723386a6b9315e947d6cbc38905e8dbd27c7c5ecd9db4525c[0m
upload image
[0;32mI (12990) esp32: DNS lookup succeeded. IP=52.219.40.104[0m
[0;32mI (12990) esp32: ... allocated socket[0m
[0;32mI (13040) esp32: ... connected[0m
[0;32mI (13190) esp32: ... socket send success[0m
[0;32mI (13190) esp32: ... set socket receiving timeout success[0m
urlTrigger activatedend rc: 0
[0;32mI (18230) esp32: Subscribe callback[0m
Subscribe callback
[0;32mI (18230) esp32: Topic: esp32/sub/data[0m
[0;32mI (18230) esp32: Detected results: [0m

assert failed: str_split main.c:515 (idx == count - 1)

```

Even after setting the Policy Action to '\*', the same error above occurred.

Active version: 4 <a href="#">Info</a>			Builder	JSON
Policy effect	Policy action	Policy resource		
Allow	iot:Connect	arn:aws:iot:ap-southeast-1:124911848185:client/myesp32-cam-example		
Allow	iot:Subscribe	arn:aws:iot:ap-southeast-1:124911848185:topicfilter/esp32/sub/+		
Allow	iot:Receive	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/sub/url		
Allow	iot:Receive	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/sub/data		
Allow	iot:Publish	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/pub/data		
Allow	iot:Publish	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/pub/url		
Allow	*	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/sub/data		

Only after allowing all Policy Actions under all Policy Resources using double wildcards '\*' did the error get resolved. Thus, we are not sure exactly which Policy Resource needs to be enabled to obtain the final label values from the MQTT topic, but allowing all Policy Actions managed to solve the problem.

<div>Active version: 2 <a href="#">Info</a></div>			<div>BuilderJSON</div>
Policy effect	Policy action	Policy resource	
Allow	iot:Connect	arn:aws:iot:ap-southeast-1:124911848185:client/myesp32-cam-example	
Allow	iot:Subscribe	arn:aws:iot:ap-southeast-1:124911848185:topicfilter/esp32/sub/+	
Allow	iot:Receive	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/sub/url	
Allow	iot:Receive	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/sub/data	
Allow	iot:Publish	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/pub/data	
Allow	iot:Publish	arn:aws:iot:ap-southeast-1:124911848185:topic/esp32/pub/url	
Allow	*	*	

## 2. ESP RainMaker Platform Setup

### 2.1 Issues During Setup

The main issue that we encountered was the different “Failed to connect to ESP32” errors. Some examples are “Invalid head of packet” and “Timed out waiting for packet header”. In our case, we just tried pressing the “FLASH” button multiple times until the connection succeeded. However, a more rational approach would have been to adjust the baud rate and see which one works better for establishing a connection with the MakePython board.

Nevertheless, the following are useful information that may help in setting up ESP-IDF and communicating with the ESP32 module:

1. On VSCode, using the ESP-IDF VSCode Extension, we should operate in an ESP-IDF terminal, instead of any usual normal terminal. This is because the environment variables are set properly only in an ESP-IDF terminal, allowing us to access the scripts under the tools directory properly (like idf.py).
2. Follow the steps specified here to establish a serial connection with ESP32: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/establish-serial-connection.html> (especially on setting the serial port device name and adding user to dialout on Linux).

### 2.2 RainMaker Architecture & Provisioning Process

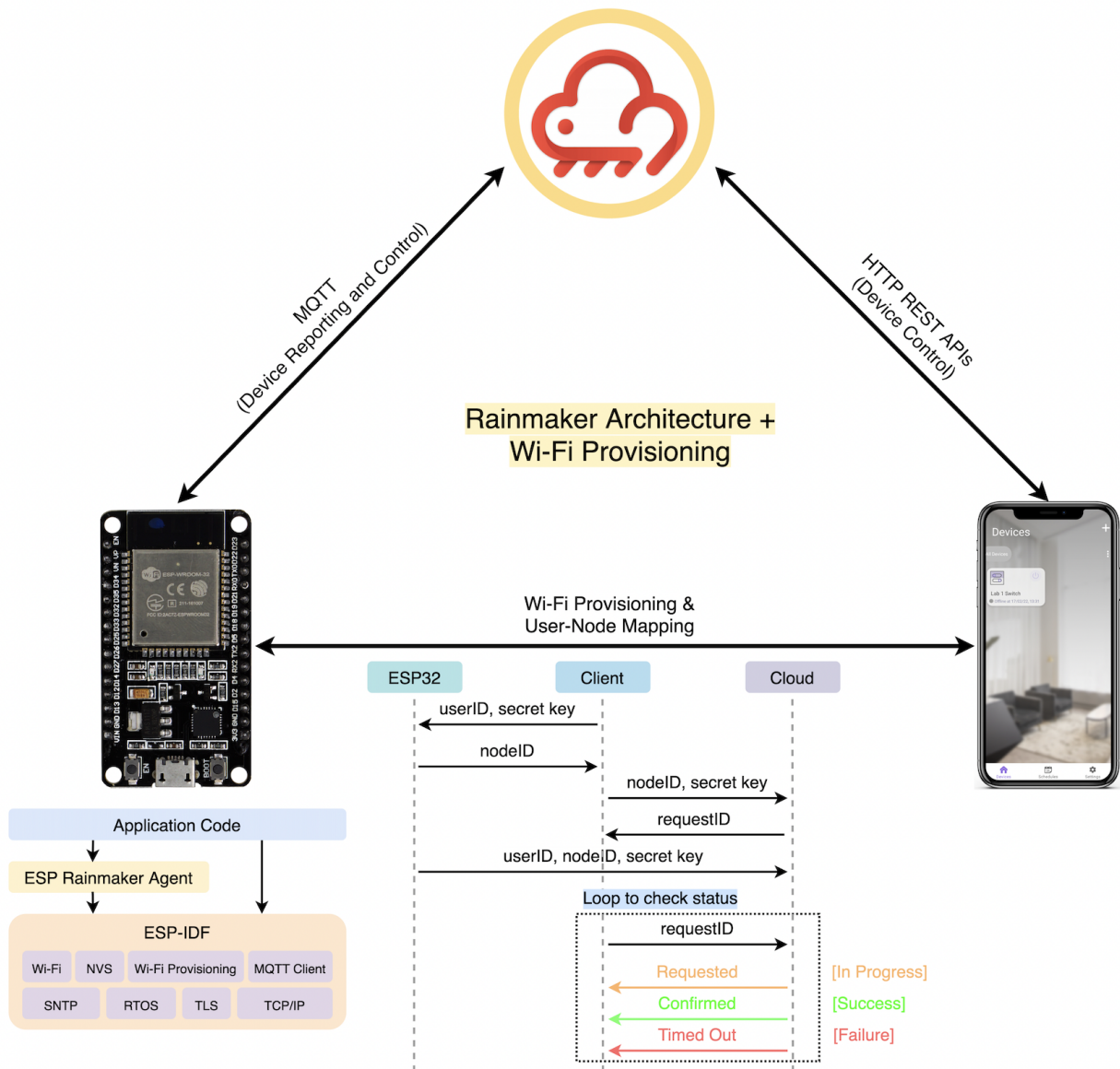
Information on RainMaker Architecture and the Provisioning Process can be found at <https://rainmaker.espressif.com/docs/get-started.html>.

The main things to take note of are:

1. ESP32 node communicates with ESP RainMaker Cloud through secure MQTT over TLS.
2. Client (phone, tablet, etc.) communicates with ESP RainMaker Cloud through HTTP REST APIs. This requires user login to ensure a secure connection with the cloud.
3. ESP32 connects to Client through User-Node Mapping that happens during the Wi-Fi Provisioning stage.

The diagram below depicts the overall RainMaker Architecture and Provisioning Process. The claiming process is not depicted in the diagram below.





### **3. Reflection on the Two Approaches**

#### **3.1 Overall Comments**

Overall, the ESP RainMaker approach provided for a much more seamless user experience as compared to the AWS IoT approach. This is because ESP RainMaker abstracts away and handles all the cloud setup automatically through the Wi-Fi provisioning stage, and also has a mobile client that allows users to control their IoT device through the phone. On the other hand, AWS IoT requires their users to log in to their web client to manually set up the necessary configurations (AWS IoT Thing, AWS Lambda, AWS S3, etc.) by themselves and to make configuration changes in the future.

However, ESP RainMaker can only be used for simple IoT applications such as configuring switches and reading sensors, while AWS Services can integrate IoT things with other applications such as machine learning/data analytics and storage services. Hence, this makes AWS Services more favourable for companies that are looking to expand their suite of capabilities to cater to a much more technologically advanced audience. AWS Services also provide better granularity and flexibility of control over the method of implementing such solutions, from the messaging protocol chosen (such as MQTT) to how collected data is stored and processed. Nevertheless, ESP RainMaker would still be a popular choice for hobbyist makers and those looking for a simple and efficient method to upgrade their homes.

#### **3.2 Comments on Cloud Services**

For AWS IoT, there is generally good integration between different services (IoT Core, Lambda, S3, etc.) and serves well as a one-stop platform that contains almost everything required for IoT setup. It also helps to abstract away the complexities of setting up backend services from scratch, which saves development time. However, AWS policies and APIs are constantly being updated and changed, which may cause issues and bugs to arise in existing code bases. Furthermore, it costs money to keep AWS services running with little transparency on the billing until it occurs.

ESP RainMaker, on the other hand, serves as a transparent cloud middleman. They use an AWS Serverless backend for device management. With one click on the client (mobile phone app in our case), the provisioning process can be completed and the device is set up and running properly. This is very convenient for building large-scale IoT solutions. However, the limitation of such an end-to-end service would be that the configurations in the cloud are set and provisioned by ESP RainMaker, and cannot be modified easily from the user's end.